An Overview of the CAPITL Software Development Environment *

Paul Adams[1] and Marvin Solomon[2]

[1] Grammatech, Inc., Ithaca, NY, USA (adams@grammatech.com)
[2] Computer Sciences Department, 1210 W. Dayton St., Madison, WI 53706–1685, USA (solomon@cs.wisc.edu)

Abstract. The CAPITL programming environment is comprised of a shared, object-oriented, versioned database, an embedded logic-based data-manipulation language, and a graphical user interface. With each software object the database stores a rich set of attributes that describe its syntax, intended semantics, and relationship to other objects. CAPITL is implemented in POL, a data model and deductive query language with elements of persistent, object-oriented and logic-based programming languages. POL is implemented in and tightly coupled with C++.

A request for a derived object consists of a partial description of its attributes. A planner written in POL searches the database for tools and sources that can be combined to create an object meeting the description. Since tools are stored in the database like other objects, plans that create tools as well as intermediate inputs are possible. A builder, also written in POL, executes plans to materialize software products. The builder verifies that existing objects are current, minimally re-applying tools as sources, tools, or system descriptions change.

After an overview of the database and the POL programming system, we outline CAPITL's logic-based approach to system modelling, illustrating it with two examples. We conclude with a status report and an outline of future directions.

1 Introduction

Large software systems are hard to build and maintain. The sheer number of components involved make the management, coordination, and storage of the components difficult. Because of the malleable nature of software, components are constantly changing. Change is not limited to source text; attributes of the source files, relationships among them, tools used to process them, and even architectures of whole subsystems change as bugs are fixed, new functionality is added, and components are re-organized. Maintaining invariants in an evolving system is a critical task for any support system [41, 48, 50].

* 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).

The CAPITL [3] project at the University of Wisconsin has been investigating a logic-based approach to software configuration management [42]. The basic thesis of our approach is that if all objects in the environment carry with them sufficiently detailed descriptions, desired software products can be described declaratively and the system can infer the process necessary to build them. To test these ideas, we have constructed a environment that tightly integrates a logic-based language with a versioned, object-oriented database. By tightly coupling the database with both an imperative object-oriented language and a declarative language, CAPITL gets the best of both worlds: declarative queries and specifications, and object-oriented extensibility and state encapsulation. This paper describes the main components of the environment and outlines how they support maintenance of large software systems.

CAPITL was designed with four principles in mind.

- **Uniformity.** All objects are represented and described in a common language.
- **Locality.** The information that describes an object and its relationships with other objects is directly associated with that object.
- **Extensibility.** New types of objects, new descriptive properties, and new relationships can be added easily.
- **Flexibility.** Policies for access control and modification to objects can be specified rather than such policies being "wired in."

CAPITL consists of three main components: a shared, versioned database, a graphical user interface, and a fully embedded logic-based data-manipulation language. The CAPITL database records all aspects of software construction: source files, documentation, sub-systems, system descriptions, tools, executables, and configurations. The form and function of each database object, as well as its relation to other objects, is described in detail by attributes stored with it. Support for efficient maintenance of multiple versions of the database is built in. A compatibility feature allows existing Unix tools to manipulate CAPITL objects as if they were Unix files.

The database is accessible via an interactive browser/editor based on the X Window System [45] and the InterViews graphical toolkit [34]. CAPITL's browser can navigate the version history of the database and the links between objects. It also provides facilities for the display and manual update of objects.

Most of the features of CAPITL are implemented in POL [3], a data model and deductive query language synthesized from elements of persistent, object-oriented, and logic-based programming languages. POL is tightly integrated with the database (all database objects are POL terms) and with a general-purpose host language (C++). The database uses the Exodus toolkit [11] to provide low-level concurrency control, error recovery, and network access. POL includes a logic-based programming language called Congress, which also servers as a query and update language.

[3] Computer Aided Programming In The Large

CAPITL uses Congress as the basis for a tool that automatically builds and maintains derived objects. An architect of a software system describes tools and policies using Congress as a specification language. A program written in Congress accepts declarative specifications of desired products and deduces plans to locate or construct them. CAPITL thus provides a platform supporting application-specific notions of consistency and correctness.

The remainder of this paper is organized as follows. Section 2 describes the POL data model, the embedded language Congress, and the interface between Congress, C++, and the database. The use of the database to store software objects is explored in Section 3. Section 4 explains how CAPITL is used for software configuration management (SCM). Section 5 illustrates these ideas with two concrete examples. Section 6 discusses related work. We close with a status report and future plans.

2 POL

POL (Persistent Objects with Logic) is a mixture of three styles of programming language: 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++ (§ 2.2), persistence from the Exodus database toolkit (§ 2.3), and logic-based features from Congress – a derivative of Prolog (§ 2.4). POL integrates these components with a shared data model and a two-way embedding of Congress in C++ and C++ in Congress. The remainder of this section describes the data model, the three components, and the interfaces between them.

2.1 Term Space

As in Prolog and LISP, POL uses one data structure for both programs and data. A *term space* is a directed graph with labelled nodes and arcs. The label associated with a node is called its *functor*[4] 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.

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

POL 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.

POL supports multiple versions of the term space called *worlds*, and uses an algorithm devised by Driscol *et. al.* [16] that supports efficient "checkpointing" of the entire term space. POL has operations to save the current term space as a world, and to reset its state to any previously saved world. A checkpoint operation does not copy the entire term space, but only an amount of data proportional to the changes made since the previous checkpoint.

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++ allows multiple inheritance and supports information hiding via explicit public/private declarations. Subclasses can override methods of their super-class as well as add new data fields and operations.

C++ classes are used in POL to provide a concrete realization of term space nodes and arcs. Data structures used to represent nodes come in a variety of flavors. Leaf nodes (nodes with no outgoing arcs) are 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, 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 Section 3.

2.3 Exodus

Exodus [11] is a toolkit for creating custom database systems. POL uses two components of Exodus, a low-level storage subsystem and a persistent programming language. The Exodus Storage Manager provides efficient access to an arbitrary-sized persistent chunk of uninterpreted data called a "storage object" through a unique identifier called an "OID." The Storage Manager supports concurrency control through two-phase locking, and a simple transaction facility with full recovery from hardware and software failures. The E programming language [43] 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 (e.g. `dbint`, `dbclass{ ... }`, etc.). Instances of a db type can be allocated from a persistent heap. The E runtime support library ensures that persistent data structures are securely stored on disk at the end of a transaction, and are fetched on demand (whenever a pointer to one is dereferenced). 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

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 [5], an extension of Prolog that supports cyclic terms. It provides transparent persistence, and has an identity-based rather than value-based semantics. The following paragraphs briefly describe the syntax and semantics of Congress. 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. In particular, 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.

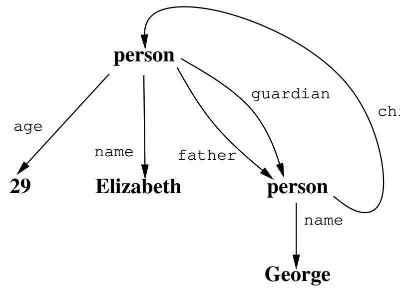
The operational behavior of Congress is defined by the same recursive backtracking search as in Prolog. A *goal* (or "query") consists of a term. It is "called" ("evaluated," "proved") by searching the procedure named by its functor for a clause whose head "matches" the goal. If a matching clause is found, each term in its body is called in turn. If no matching clause can be found, the interpreter backs up by undoing all of its actions since the last "choice point" (the point at which a clause was chosen to match against a goal) and attempts another match. The process continues either until all goals and subgoals have been proven, 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*[5] Congress uses a variant of unification that supports cyclic terms [5]. 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

[5] Background material on unification can be found in many logic programming texts and in an excellent survey by Knight [28].

successful unification is an equivalence relation that records which nodes were matched. The evaluation of a call adds a *copy* of a clause to the term space and identifies nodes matched as a result of unifying its head with the query. The terms of the body are called in this extended term space.

Congress has a character-string *expression language* 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(`$\mathtt{s}_1$ $=>$ $\mathtt{t}_1$`, ..., `$\mathtt{s}_n$ $=>$ $\mathtt{t}_n$`)`, where `f` is its functor, $\mathtt{s}_1$`, ..., `$\mathtt{s}_n$ are the selectors of the arcs with `f` as their tail, and $\mathtt{t}_1$`, ..., `$\mathtt{t}_n$ 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[6]

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

The expression language denotes a clause with head $\mathtt{t}_0$ and body $\mathtt{t}_1$`, ..., `$\mathtt{t}_n$ as "$\mathtt{t}_0$ `:-` $\mathtt{t}_1$`, ..., `$\mathtt{t}_n$." The expression language also includes "syntactic sugar" for representing common infix operators such as `+`, `*`, `-`, `/`, and for Prolog style 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 occurrences of `@` can be omitted in most cases. For example, `f(a,X)` is the same as `f(1=>a, 2=>X:@)`.

With these abbreviations, the Congress expression language becomes a strict superset of Prolog. It 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[7]. For example, the Congress expression `employee(age=>25,salary=>30)` is less confusing than the corresponding

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

[7] It also has a rather subtle effect on the definition of unification. See the LOGIN paper [5] for details.

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. The expression language of Congress uses the functor "`@`" for the first purpose and tags for the second.

2.5 Embedding

The coupling between C++ and Congress is a two-way embedding: Each language appears to be an embedded sub-language [37] of the other. Each language retains its own style. The embedding does not alter the syntax or semantics of either language. Since Congress programs are C++ data structures, a C++ program can construct or modify a program and call the Congress interpreter to execute it, capturing all output as C++ structures. The embedding is bidirectional: C++ procedures can be declared as *external predicates* in Congress. When the interpreter encounters a goal whose functor is an external predicate, it calls the corresponding C++ procedure, passing it the goal and a description of the current state of the computation (added nodes and bindings). The procedure may make any modifications to the environment it deems appropriate (for example, adding bindings) and return either a success or failure indication. During backtracking, the interpreter may call the procedure again, asking whether it can succeed in other ways. In short, an external predicate is any C++ procedure that follows the protocol of a Congress procedure.

External predicates have proven extremely useful. They are used to implement all of the built-in predicates usually found in Prolog implementations, such as arithmetic operations, as well as other functions that are awkward or impossible to implement directly in Congress, such as file system access or invocation of other programs. The Congress interpreter is itself an external predicate so Congress programs can invoke the interpreter recursively.

3 CAPITL Object-base

CAPITL uses the persistent term space of POL to build an object-oriented database (or *object-base* for short). All the entities used during the process of software development – source files, derived binaries, documentation, executable tools, and descriptions of subsystems – reside in the object-base. Properties attached to each object describe it and its relationship to other objects. The object-base is organized into a tree-structured naming hierarchy similar to a Unix file system (§ 3.1). The object-base can be accessed interactively, from programs written in C++ or Congress, or through a Unix compatibility feature (§ 3.3). In the last case, an extension to Unix path-name syntax provides access to versions of the term space.

3.1 Objects

The fundamental entity in the CAPITL database is the *object*. An object is a "heavier weight" term that is guaranteed to have certain selectors with built-in semantics. Viewed from the Congress language, the term space is simply a labelled directed graph. Viewed from C++, the nodes of the graph are further classified into an inheritance hierarchy. As explained in § 2.2, the first level of the hierarchy separates nodes into leaf nodes (which are further classified as integers, byte strings, etc.) and internal nodes, which contain pointers to other nodes. Among internal nodes, CAPITL further designates some as *object* nodes, which implement the semantics of CAPITL objects.

A CAPITL object can be viewed as a POL term (*i.e.* a directed labeled graph) and it supports the interface of the POL `Term` class. Nevertheless, a CAPITL object is best viewed as a set of <*name, value*> pairs, called *attributes*, where the value is a POL term. As C++ class instances, CAPITL objects have methods for manipulating their associated data. The main methods for manipulating CAPITL objects relate to attributes; the POL term interface for objects is created using these methods.

CAPITL distinguishes three kinds of attributes: *simple*, *timestamped*, and *derived*. A simple attribute is a pairing of a name with a term and behaves exactly like a POL term. A timestamped attribute records when its value was last modified and provides a method to retrieve this timestamp. The value of a derived attribute is not represented directly as a term, but as a function application. CAPITL maintains a cache of the most recent value of a derived attribute. The inputs to a derived attribute are timestamped attributes, so that invalid cached values can be detected and only recomputed when necessary.

The value of a derived attribute is represented by a special node called a *function* node. These nodes replace all the read operations of the term interface with code that first validates the cached function result and then performs the requested operation by invoking the corresponding method of that value. Function nodes are read-only; all write operations of the term interface are overridden by operations that return a failure indication.

All objects have simple, integer-valued attributes `owner`, `group`, `permissions`, `mtime`, `atime`, and `ctime`, interpreted as in Unix. Objects also have a *directory* attribute containing a reference to the directory in which the object resides. The primary value of an object is stored in the *contents* attribute whose representation depends on the kind of object.

Objects are further classified as *directories*, *files*, and *symbolic links*. A directory object is similar to a Unix file-system directory. Its `contents` attribute is a list (constructed of `cons` nodes) of references to other objects. The directories create a tree-structured name space similar to the Unix file system.

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 (§ 3.3). 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 [49].

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.

Finally, *symbolic link* objects exist to support Unix compatibility interface (§ 3.3). The `contents` attribute of a symbolic link is a printable-string atom.

3.2 Versions

CAPITL uses the world 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. The last operation behaves as if it were making a complete copy of the parent database state, but is much more efficient. A world can also be unfrozen if it has no children. A modifiable (leaf) world of the database may be thought of as a "workspace." A person who wishes to modify the database generally selects an existing committed world, creates a modifiable world derived from it, and makes the modifications in the new world. When the changes have reached a stable state, the new world may be committed. Policies and mechanisms for mediating shared access to modifiable worlds are still under study.

Each world has a unique *version ID*, which is a non-empty sequence of positive integers. The root world has ID "0". The ID of the first child of a world W is formed by incrementing the final component of W's id. Sibling worlds are formed by appending zeros to W's ID. For example, the children of world 1.3.2 would be labeled 1.3.3, 1.3.2.0, 1.3.2.0.0, etc. This numbering is similar to the scheme used by RCS and SCCS, and seems more natural than "Dewey decimal" numbering in the common case of long sequences of single-child worlds. For example, a sequence of consecutive derivations from 1.3.2 would yield 1.3.3, 1.3.4, 1.3.5, etc. This numbering scheme can, however, become quite confusing when multiple worlds are derived from the same parent. We expect that worlds will normally be selected by symbolic name or other attributes stored in an index structure (itself stored in the database) rather than by version ID. (This part of the database is still under development.)

In general no changes are permitted in a committed world. However, the value of a derived attribute may be safely deleted and replaced by the special atom "not available." Switching the state of a derived attribute between available and not available is considered a "benign" modification of the database and is permitted in committed worlds.

3.3 Accessing a CAPITL database

A CAPITL database can be accessed and manipulated in several ways:

– Directly, through programs written in C++ or Congress.
– Through an interactive X-based browser.
– Through a Unix-compatible interface called EFS.

CAPITL is written in POL, so all of its structures can be accessed as data structures in C++. For example, nodes are all instances of the class `Term`, which exports such methods (member functions) as

```
boolean IsLeaf();
```

which enquires whether the term is a leaf, and

```
Term *Edge(char *selector);
```

which returns the term referenced by a particular selector (if one exists). Class `Integer` is a subclass of `Term` with an `IntVal()` method that returns its integer value, and so on. Documentation for this interface is currently being written [3].

Browser. An interactive browsing interface has been written on top of the X Window System using the InterViews [34] toolkit. The browser supports visiting any object or directory in the object-base and uses type-sensitive displays to depict the `contents` attribute of an object; other attributes are displayed using the Congress expression language (§ 2.4). For example, the `contents` attribute of a source file object is displayed using a text viewer in the *Viewing* box and any other attributes in the *Term View* box (see Figure 1). The current focus can be moved to a neighboring object in the naming hierarchy by double clicking an object named in the *Object Path* or *Object Siblings* box. The focus can also be changed by typing a path name in the *Location Selector* box. "Time travel" is accomplished by typing a version ID in the *Current World* box. Menus exist for creating and destroying objects, invoking the Congress interpreter, and for creating and committing worlds. Multiple simultaneous windows on a database are kept consistent with changes made from any of them.

EFS. 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. However, the differences between the CAPITL object base and the Unix file system interfere with using existing tools. Consider, for example, the problem of compiling a C source file stored in CAPITL. One approach is to copy the `contents` of the source object into an ordinary Unix file, invoke the compiler, and copy the resulting object module back into CAPITL. A second approach is to store in CAPITL "stub" objects that contain pointers (path names) to Unix files. A third approach is to modify the compiler (perhaps by linking it with alternative versions of the Unix library functions `open`, `read`, `seek`, etc.) so that it can read and modify CAPITL objects. None of these approaches is entirely satisfactory.

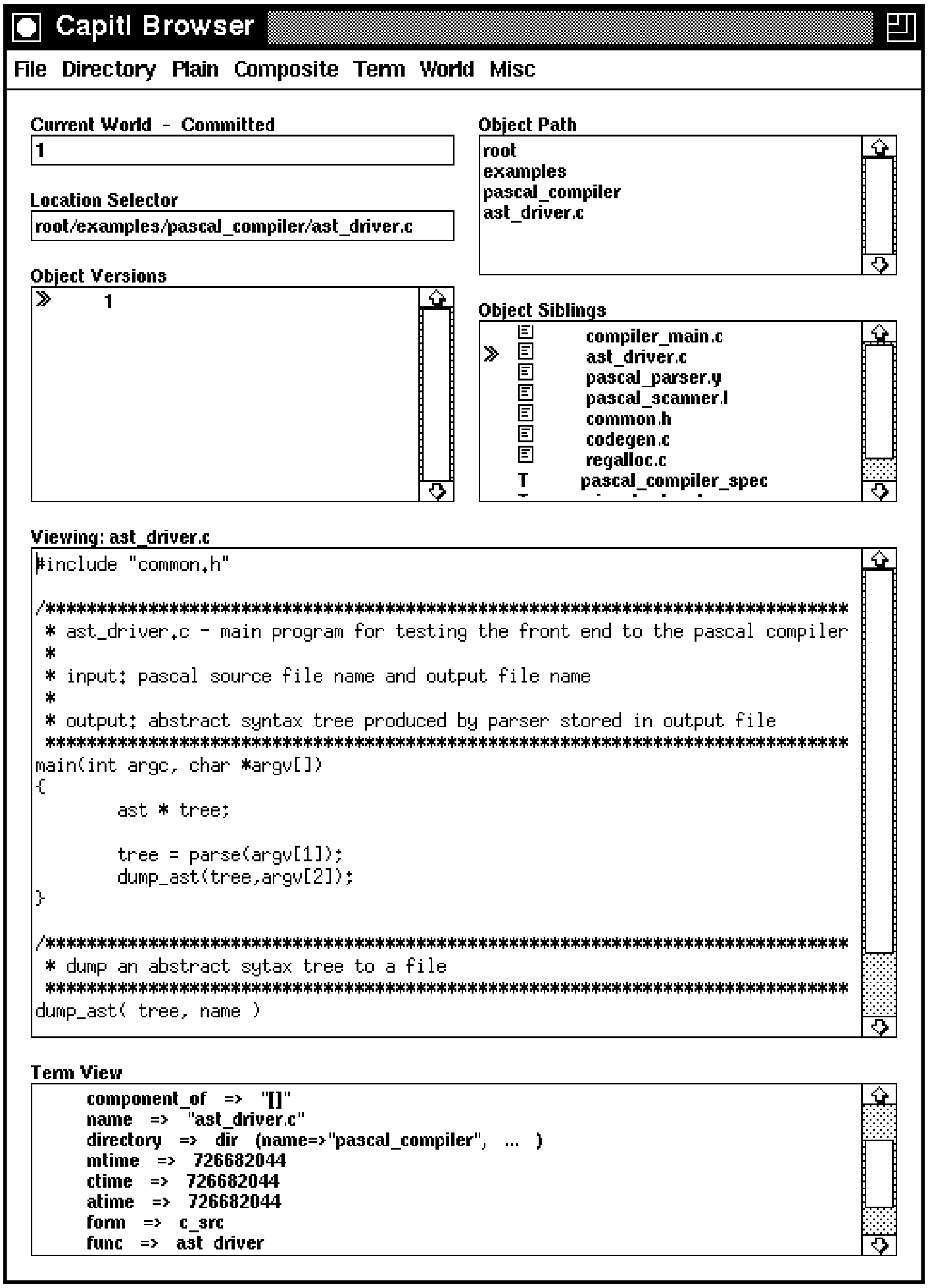


Fig. 1. CAPITL Browser

The *Emulated File System* (EFS) allows programs to access CAPITL objects as if they were Unix files. It is based on the Network File System (NFS) facility [44], which is included in most versions of Unix. NFS was originally designed to support transparent access to remote files. A version of the `mount` command associates a remote file system with a name, called a *mount point* in the local file system. System calls that request operations on files below this mount point are forwarded to an NFS server. Normally, the server is the Unix kernel on the remote system, which executes the requests on its local disk. The remote file system thus appears to be grafted into the local name space as a subtree of the mount point. It is possible, however, to designate a user-level process as an NFS server (Figure 2).

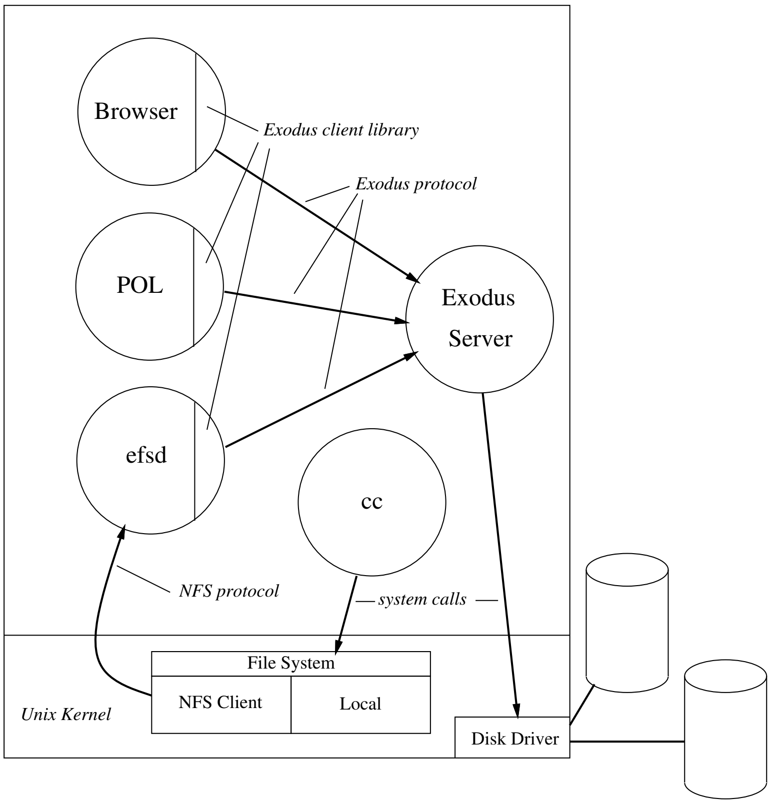


Fig. 2. CAPITL Process Architecture

The EFS daemon *efsd* emulates a Unix file system on a CAPITL database. Once a CAPITL database is mounted, its objects can 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.

Not all features of the CAPITL database are accessible through EFS. For example, a composite or term object appears to be an empty file from EFS (it behaves like `/dev/null`). However, EFS does allow access to alternative worlds through an extension of Unix path name syntax. A version ID followed by a colon is interpreted as a request to resolve a path name in a designated world of the database. Path names without version ID's are resolved in a *current world* analogous to the current working directory. For example,

```
diff 3.3:prog.c prog.c
```

compares version 3.3 of `prog.c` with the current version, and

```
(echo -n "updates done "; date) >> 3.5:log
```

adds a line to version 3.5 of `log`.

As in Unix, a path name that does not start with "/" is interpreted relative to the current directory (and world). Since the Unix kernel uses the same mechanism to resolve `chdir` requests as `open`, the shell's `cd` command can be used to navigate among worlds. For example,

```
cd 3.2.1:
```

sets 3.2.1 as the default world for subsequent file-system requests. The path name supplied in a *mount* request is interpreted in the same way, so a default world can be specified at mount time, as in

```
efs_mount /3.4:@hostname project.old
efs_mount /3.5:@hostname project.new.
cd project.new/include
vi defs.h
```

Although EFS was created for CAPITL, it can be used by any C++ system that needs to provide a Unix-compatible interface. It is packaged as a driver program and a set of abstract C++ classes that encapsulate that NFS model. Application-specific classes inherit from these classes. For example, the CAPITL `Directory` class is derived from both `Term` and `EFSdir`. Class `EFSdir` declares methods to add, delete, and lookup directory entries, but supplies no implementation (they are "pure virtual member functions" in C++ terminology). The CAPITL `Directory` class implements these functions by manipulating the list stored in the `contents` attribute of the term. More details about the EFS package will be contained in a forthcoming report [46].

4 Software Configuration Management in CAPITL

CAPITL provides assistance for constructing and maintaining software products. A product is the output of a series of tools applied to a set of objects. The goal is to produce a "correct" version of a product as efficiently as possible.

Correctness is a fuzzy concept that has possibly different meanings in different situations. Therefore, CAPITL supports a powerful constraint language (POL) allowing system designers to describe attributes and integrity constraints as appropriate to each application. CAPITL guarantees that all derived objects are correctly created by applying tools to inputs in accordance with these specifications. In addition, CAPITL ensures that all products are current (that is, none of the objects used to create the product has changed since the product was constructed). When building (or rebuilding) a product CAPITL speeds up the process by reusing previous work when possible.

CAPITL software objects are classified as *source* or *derived.* A source object is created by a human being or by some other process outside the control of CAPITL. A derived object is created by applying a *tool* object to an input object. Each tool has exactly one input, which can be a set or list of other objects. (Construction of such composite objects is described more fully below.) The tools as well as inputs can be either source or derived. Each derived object can be viewed as the value of a *derivation* – an expression tree whose leaves are source objects and whose interior nodes represent occurrences of a built-in *apply* operator. Objects are all represented in CAPITL as POL terms, and thus can have a variety of attributes. Some of these attributes constrain the set of well-formed derivation expressions (§ 4.1). In particular, each tool object has an `in` attribute that must match the object to which it is applied, and an `out` attribute that constrains the attributes of the result. Since the attributes are terms that can contain variables, an object can be partially specified. An incomplete object can be more fully specified by replacing occurrences of variables with other terms[8]. An object with an unspecified `contents` is called an *abstract object.*

A user requests the derivation of a new object by constructing an abstract *goal* object and invoking the *planner* (§ 4.4) and the *builder* (§ 4.5) which are Congress programs supplied with CAPITL. The planner determines how to create objects and supplies them with `code` attributes, while the builder creates the `contents` of planned objects by evaluating expressions in their `code` attribute. The `code` attribute of an object is a list of alternative *build expressions*, each of which contains references to a tool object and an input object. When asked to plan an object without a `code` attribute, the planner searches the database for tools and matching inputs such that the result of applying the tool to the input yields the desired goal. It then recursively plans the tool and the input.

The planner can infer the need to construct new derived objects. If it finds a tool capable of creating a desired goal object but no corresponding input, it tentatively creates an abstract object representing the input and plans it. If the input is a composite object (for example, a list of inputs to the linkage editor), the planner searches for a *template* (§ 4.2) that describes potential collections. The planner can also create new tool objects. A *tool description* (§ 4.3) contains an input/output description of a potential tool. If the planner fails to find a tool that can create a goal object but discovers an appropriate tool description, it

[8] A consequence of the definition of unification in § 2.4 is that the absence of an attribute is equivalent to an attribute whose value is a variable.

will instantiate the description as a new tool object and attempt to plan it. For example, the planner may fail to find a cross-compiler capable of building an object module for a particular machine architecture, but a tool description may suggest that such a compiler can be built from existing sources.

The builder completes the process of making an abstract goal concrete by executing one of the build expressions in its `code` attribute. If the build is successful, the result is placed in the `contents`, and a record of the specific tool and input used to create it is placed in the `provenance` attribute of the derived object. If it is unsuccessful (for example, if the tool is a compiler that discovers an error in its input), the build expression is marked as "failed," the planner is invoked to find other alternatives, and another build expression is tried.

4.1 Object Descriptions

The attributes `code`, `contents`, `provenance`, `form`, `functionality`, and `references` are used by CAPITL for the purpose of planning and building. The first three of these have been briefly described above. The attributes `form` and `functionality` jointly describe the "type" of an object. These attributes control the matching of tools to inputs and outputs. The `references` attribute is used to record extra dependencies among objects.

Form. The `form` of an object is its type when used as an argument to a tool. In the simplest case the `form` attribute is a simple atom with an application-defined meaning, such as `c_source` or `object_code`. More detailed information can be specified by more complicated terms. For example, the `form` attribute of an object module might be specified as

```
object_code( opt => no, debug_symbols => yes)
```

Since forms are terms, the partial order induced by unification can be used to express subtype relationships. For example, this term is compatible with (unifies with) `object_code`. The planner interprets two values for the `form` attribute as having special meaning: The value `bag(T)` is interpreted as a homogeneous multiset of type `T`, and the value `record(` $\mathtt{T}_1$`, ... ,` $\mathtt{T}_n$`)` as a heterogeneous collection of types $\mathtt{T}_1$ through $\mathtt{T}_n$. All other values are interpreted as non-structured (atomic) types.

Functionality. The `functionality` attribute of an object is a description of what the object does. For tool objects, the `functionality` attribute is the type signature of the tool and describes its behavior via an in/out pattern. For example, the `functionality` of a C compiler might be specified by the term

```
func(in => obj(form => c_source, functionality => F),
     out => obj(form => object_code, functionality => F)),
```

which states that the input must be C source code, the output will be object code, and the functionality (semantics) of the module is preserved.

Contents, Provenance, and Code. For atomic objects the `contents` attribute is uninterpreted by CAPITL. For composite objects the `contents` is the set of sub-objects that comprise the object. The `provenance` of a derived object specifies the tool and input used to create the `contents`. Since the `provenance` names specific objects (by their OID), it is completely synonymous with the `contents`, in the sense that the same `contents` could be recreated by rerunning the same tool on the same input. The `code` attribute is a set of expressions, each of which is type-compatible with the object and is thus a potential recipe for generating the `contents`; it represents a non-deterministic program for creating the `contents.` The `code`, `provenance`, and `contents` are in some sense all manifestations of the same value where the `code` is the least specific and the `contents` the most specific.

References. The `references` attribute identifies objects that are referred to inside an atomic object's `contents`. It can be an assertion from the specification writer or the output of some language-specific tool. For example, suppose a C source file has many "`#include`" preprocessor directives. If the C compiler and preprocessor are being modelled as one tool, the `references` attribute would represent the set of include files required to run the compiler/preprocessor tool. These dependencies are needed for maintaining consistency after changes. References are used in checking whether derived objects are up-to-date. They are also useful for browsing source objects.

When describing an object, a specification writer has two ways to refer to other objects. A *generic reference* is a pattern that matches other objects in the object-base. For example, the term

```
obj(functionality =>
      func(in => obj(form => c_source, functionality => F),
           out => obj(form => object_code, functionality => F))
   )
```

represents a generic reference to *any* C compiler. If the environment contained two C compilers, such as "cc" and "gcc," either could match. A *specific reference* defines a unique object. A specific reference can be created from a generic reference by adding enough other attributes to make the reference unique or by using the object's identity. Because it is difficult to guarantee that a set of attributes always identifies exactly one object, the identity of an object is preferred for specific references. For example, the `provenance` of an object module might contain `tool => Gcc` where `Gcc` is a tag for a specific compiler object.

4.2 Templates

A system is a collection of objects that when combined form a meaningful "chunk". Systems are described in CAPITL by specially formed objects called *templates.* (Such descriptions are often referred to as system models.) A template consists of a set of generic references, specifying (at least) the functionality of each sub-object, and a set of constraints. The constraints are used to specialize the generic references prior to *instantiating* the template. Templates are instantiated by making a copy of the template, calling all the predicates listed in the constraints, and then resolving each generic reference via a database lookup. The result is an object that unifies with the template, but has all the generic references in its `contents` resolved into specific references. Because template instantiations are objects, they can be used as the sub-parts of other template instantiations. Thus, system descriptions in CAPITL can be composed.

4.3 Tool Descriptions

CAPITL uses separate tool descriptions to specify the available set of tools. Tool behavior (as opposed to tool objects) is described via specially formed Congress rules stored in a separate rule-base that can be imported into a Congress program. Tool descriptions serve two purposes. First, the presence of a tool description alerts the planner that it is possible to build a tool with a particular functionality. Hence tool generators can be modelled in CAPITL. Second, tool descriptions allow more complicated descriptions of tool behavior than is possible with simple input/output signature patterns. Section 5.2 contains an example that illustrates this feature.

4.4 Planning

Planning is the process of finding a set of source objects and tool applications needed to satisfy a request. The planner makes use of three kinds of data: existing planned objects, template objects, and tool descriptions. The planner searches the space of all possible "well formed" tool and object combinations for expressions whose results match the goal and that only contain references to atomic objects or fully specified composite objects. These expressions are stored in the `code` attribute of an object and represent a *potential* recipe for constructing the `contents` of that object – the planner does not guarantee that an expression will succeed when evaluated. During planning, templates are instantiated and any constraints attached to objects are checked ensuring that all objects used to build a system conform to the constraints specified in the description of the system. Such constraints can easily encode certain kinds of semantic correctness such as "use all debugging versions," by forcing the appropriate sub-types to be used for derived objects (and hence forcing the use of tools that produce debugging information).

The planner avoids repeating work by keeping track of the current state of an object's plan with an additional `plan_state` attribute. When the planner creates a new object to represent an intermediate result, its state is *attempted.* If a build expression is found, the state changes to *successful*, otherwise it changes to *failed.* Using this attribute the planner can avoid attempting to plan an object that has already failed and avoid derivation loops, which occur if an object is needed in order to derive its own `contents`.

Planning may be computationally expensive because it exhaustively searches a potentially exponential space. To speed the planning process we are experimenting with lazy generation of expressions and better search strategies. Instead of deriving all equivalent expressions, our prototype stops as soon as a single expression is found. (A "replan" request is available to search for additional expressions.) Currently, the planner uses a blind depth-first search. By ranking choices, perhaps by using approximate tool costs, a branch-and-bound strategy could be used to improve performance.

4.5 Building

Given a reference to a derived object, the job of the builder is to make the `contents`, `provenance`, and `code` attributes consistent. They are consistent if the `provenance` is one of the expressions in the `code` and it evaluates to the `contents`.

The builder traverses the expression graph defined by the `code` and `provenance` attributes. Depending on the state of an object there are three cases that arise:

1. The object does not have a `provenance` or `contents` attribute. The builder evaluates the first untried expression in the `code`. If the expression is successful, the builder stores that expression as the `provenance` of the object. If the expression fails (perhaps due to a syntax error in a source program), the builder annotates that expression as unsuccessful and invokes the planner, which attempts to find a different expression for the object. If the planner is successful, the build continues.
2. The object has a `contents` (and hence a `provenance`). In this case the builder must determine if the `contents` is still valid. In a committed world, the `contents` must be the value of the expression stored in the `provenance`, so no reconstruction is necessary. In a mutable world, objects referenced by the `provenance` may have changed since it was last evaluated. To determine if the `contents` is still up to date, the builder uses timestamps. The `mtime` attribute of an object records the time its `contents` was last modified. Associated with the `provenance` of a derived object is a `provenance_timestamp` that records the timestamps of the tool and argument. If the timestamps match, the `contents` is still valid. Otherwise, the `provenance` is re-evaluated. A more precise notion of validity that relies on semantic properties of the objects involved could be used, potentially allowing fewer expressions to be evaluated [51].
3. The object has a `provenance` but no `contents`, indicating that a user conserved space by deleting the `contents`. The `contents` can be regenerated by simply evaluating the `provenance`.

4.6 Discussion

Separating the planning and building phases has several advantages. The decomposition of a system into sub-systems tends to change slowly, allowing the output of the planner to be used many times. Hence the cost of planning is amortized over many builds. Moreover, the separation simplifies the builder: It is only concerned with equivalence between a functional expression and a "cached" copy of the result of that expression; the planner does all the work of selecting objects and tools.

One can view the planner as a code generator and the builder as a code evaluator. To increase the speed of building a separate optimize phase could be used after planning to perform traditional compiler optimizations such as strength reduction and elimination of intermediate values. These optimized expressions would then be saved in the `provenance` for future use. For example, the expressions produced by the planner uses an object to hold the result of every tool application. A linear sequence of tool applications in which the intermediate results were not specifically requested could be compressed into a single pipeline invocation.

CAPITL worlds provide a means to group objects with similar semantic properties. This version mechanism assists planning by limiting the search needed to construct a product. Because only one version of an object is visible in a given world, the planner does not need to choose among the (potentially large) set of versions of each source object. Hence the combinatorial explosion associated with combining components represented as version sets is avoided during planning.

5 Examples

To illustrate the concepts of the previous section, we present two examples of simple subsystems. The first example illustrates how an executable program is built from sources in a variety of languages. The second example is drawn from the domain of document processing.

5.1 A Pascal Program Analyzer

The first example is a simple program analyzer that translates Pascal source files into abstract syntax trees (AST's). Such an analyzer might be a component of a compiler or other larger system. We assume that four source objects are available: a Lex [32] specification of tokens, a YACC [22] specification of a grammar, a driver program written in C, and a common file of declarations included by all three sources. These objects are shown in Figure 3. `Common` is the file of common declarations. It has only two attributes, a format (source for the C preprocessor) and the actual text contents. `Main` is the main program. Its format is C source, it depends on `Common`, and its functionality (semantics) is described by the atom "`Cast_driver`." The Lex and YACC source objects are similar. The `references` attributes might be supplied manually by the author of the program or it might be deduced by a tool such as the Unix `makedepend` utility. The `functionality` attributes would be supplied manually by the designer of the analyzer package.

`Pascal_Analyzer` describes the functionality of the desired tool: It should translate a Pascal source object into an abstract syntax tree preserving its func-

```
Common:obj(
    form => cpp_include,
    contents => "#include <stdio.h>; ..."
).
Main:obj(
    functionality => ast_driver,
    form => c_source,
    contents => "main(int argc, char *argv[]) ..."
    references => [ Common ]
).
Scanner:obj(
    functionality => pascal_scanner,
    form => lex_source,
    contents => "..."
    references => [ Common ]
).
Grammar:obj(
    functionality => pascal_parser,
    form => yacc_source,
    contents => "..."
    references => [ Common ]
).
Pascal_Analyzer:func(
    in => obj(form => pascal_source, functionality => F),
    out => obj(form => ast, functionality => F))
).
Analyzer_Spec:obj(
    functionality => Pascal_Analyzer,
    form => bag(type => T),
    contents => [
        C1:obj(functionality => ast_driver, form => T)],
        C2:obj(functionality => pascal_scanner, form => T),
        C3:obj(functionality => pascal_parser, form => T),
    constraints => [ debug_level(C1), debug_level(C2), debug_level(C3) ]
).
```

Fig. 3. Source Objects

tionality. `Analyzer_Spec` is a template that specifies how the component functionalities `ast_driver`, `pascal_parser`, and `pascal_scanner` can be assembled to produce a tool that translates Pascal source into abstract syntax trees. `Analyzer_Spec` may be thought of as a "tool" that produces a composite object (a package of objects) from components. This specification conveys three pieces of information: First, that the functionality `Pascal_Analyzer` is the sum of the functionalities `ast_driver`, `pascal_parser`, and `pascal_scanner`; second, that resulting object has form `bag(T)`, where `T` is the form of each component; and third, that all the components should have a property called `debug_level`. For example, if it desired that all object files have the `debugging` property, the `debug_level` predicate would be defined as

```
debug_level(obj(form => F:object_code)) :-
    !, F = object_code(debug_symbols => yes).
              % object modules must have debugging information
debug_level(obj).  % all other objects pass the test vacuously
```

Figure 4 shows a variety of tool specifications. The tool specification `Lex_Spec`

```
Lex_Spec:tool(
    functionality => Lex_Func:
        func(in => obj(form => lex_source, functionality => F),
            out=> obj(form => c_source, functionality => F))
).
Lex:obj(
    form => executable,
    functionality => Lex_Func,
    contents => "..." % the actual executable code
).
Yacc:obj(
    form => executable,
    functionality =>
        func(in => obj(form => yacc_source, functionality => F),
             out=> obj(form => c_source, functionality => F)),
    contents => "..."
).
Cc_debug:obj(
    form => executable,
    functionality =>
        func(in => obj(form => c_source, functionality => F),
             out=> obj(form => object_code(dbg_sym => yes, opt => no),
                       functionality => F)),
    contents => "..."
).
Cc_opt:obj(
    form => executable,
    functionality =>
        func(in => obj(form => c_source, functionality => F),
             out=> obj(form => object_code(dbg_sym => no, opt => yes),
                       functionality => F)),
    contents => "..."
).
Ld:obj(
    form => executable,
    functionality =>
        func(in => obj(form => bag(type => object_code),
                       functionality => F),
             out=> obj(form => executable, functionality => F)),
    contents => "..."
).
```

Fig. 4. Tool Objects

describes the functionality of a Lex processor – it transforms a `lex_source` input into `c_source` preserving semantics. The object `Lex` is an executable program that conforms to this specification. Similarly, each of other tools would have a corresponding tool description. Since each tool description is identical to the input/output signature of the corresponding tool, we omit the remaining descriptions. We have chosen to model the debugging and optimizing versions of the C compiler as two distinct tools.

Calling the planner with the goal

```
obj( form => executable, functionality => Pascal_Analyzer )
```

will create a partial object with the desired functionality and form. If planning is successful, the object will contain a build expression in its `code` attribute that can be used to create its `contents`. The planner will also create partial objects for intermediate objects as shown in Figure 5. At this point, the builder may be

```
Goal:obj(
    functionality => Pascal_Analyzer,
    form => executable,
    code => [ build_expr(expr => apply(Ld, Object_Modules, Goal)) ]
).
Object_Modules:obj(
    functionality => Pascal_Analyzer,
    form => bag(type => object_code),
    contents => [ ScannerObj, ParserObj, MainObj]
).
ScannerObj:obj(
    functionality => pascal_scanner,
    form => object_code(debug_symbols => yes, opt => no),
    code => [ build_expr(expr => apply(Cc_debug, ScannerC, ScannerObj)) ]
).
ScannerC:obj(
    functionality => pascal_scanner,
    form => c_source,
    code => [ build_expr(expr => apply(Lex, Scanner, ScannerC)) ]
).
ParserObj:obj(
    functionality => pascal_parser,
    form => object_code(debug_symbols => yes, opt => no),
    code => [ build_expr(expr => apply(Cc_debug, ParserC, ParserObj)) ]
).
ParserC:obj(
    functionality => pascal_parser,
    form => c_source,
    code => [ build_expr(expr => apply(Yacc, Grammar, ParserC)) ]
).
MainObj:obj(
    functionality => ast_driver,
    form => object_code(debug_symbols => yes, opt => no),
    code => [ build_expr(expr => apply(Cc_debug, Main, MainObj)) ]
).
```

Fig. 5. Derived Objects After Planning

invoked on the object `Goal`. Assuming there were no errors, the builder would fill in the `contents` and `provenance` attributes of each object in Figure 5.

Suppose the state of the system is frozen by committing the current world and a new one is created. Consider two different kinds of modifications to the system:

1. The only action in the new world is to modify the `contents` of `Scanner`. In this scenario, the same plan can be reused (in fact calling the planner will result in no changes), and the builder will reuse `ParserObj` and `MainObj`, rebuilding only `ScannerC`, `ScannerObj`, and `Goal`.

2. No sources are modified, but an optimized version of the analyzer is desired. The definition of the Congress predicate `debug_level` is changed to

   ```
   debug_level(obj(form => F:object_code)) :-
       !, F = object_code(opt => yes).
   debug_level(obj).
   ```

   There are two choices for where to store the new version: in a new object distinct from the existing debugging version or in the existing analyzer object (replacing the debugging version in this world). In the latter case, the `code` attribute would need to be cleared before planning (if not, the object would fail the current set of constraints and would fail to be used). In either case the planner would be re-invoked on the appropriate goal object. The existing object `Object_Modules` would not be used because its constraints fail with the current definition of `debug_level`. Instead, the planner would create a new composite object from the `Analyzer_Spec` template that uses the output of `Cc_opt` rather than `Cc_debug`. The derived objects `ParserC` and `ScannerC` from the previous world would be used without modification.

5.2 Document Processing

The Unix *troff* document processing system includes a variety of special-purpose preprocessors. If a document does not use a particular feature, the corresponding preprocessor need not be applied. For example, *eqn* only needs to be run on documents that contain mathematical equations, while *tbl* is only required for documents containing tables. We could model these tools by defining a variety of types as in the previous example, defining *eqn* to be a translator from `troff_with_eqn` to `troff`, etc. However, this approach would require a different type for each subset of the features. A better approach defines one form, `troff`, with subtypes for different sets of required features. For example a document with equations and tables would have form `troff(features => [eqn, tbl])`. The tool description for *eqn* would then be

```
tool(functionality => func(
     in => obj(form => troff(features => L1), functionality => F),
     out=> obj(form => troff(features => L2), functionality => F)))
       :- delete_feature(feature => eqn, in => L1, out => L2) ,
```

where the predicate `delete_feature` searches list `L1` for an occurrence of `eqn` and removes it. If a document does not use *eqn* the body of this rule will fail and the planner will not consider it applicable. A more sophisticated version of `delete_feature` can encode the requirement that some processors have to be run before others. As with the `references` attribute, the `features` of a *troff* document could be added manually or by a processor that analyzes a document.

6 Related Work

Many of the ideas in the design of CAPITL are present in other systems. CAPITL's main distinguishing feature is tight integration of a process programming language (Congress) with an underlying versioned, object-oriented database. A software object is not just described by a Congress expression, it *is* a Congress expression. Congress is declarative rather than procedural; it allows system designers to concentrate on the properties of objects and subsystems rather than on procedures to manipulate them.

All of the following systems provide good support for building; they differ primarily in how systems are specified, how version selection is accomplished, and what kinds of consistency are guaranteed.

The related work in this section is organized into three sections: build tools, environments that integrate building, versioning, and other aspects of software development, and work that uses logic as the basis for SCM or building.

6.1 Build Tools

Many successors to Make [20] have been built that keep the basic idea of a Makefile for describing systems. Each of the newer systems addresses deficiencies in Make and adds new features that make it easier to use. There are many variations of these so-called "super makes" and we describe only a sample here. Tools that are comparable to Make are included here as well.

GNU Make. The Make utility from the Free Software Foundation [47] enhances the original Make in several ways. (The authors give credit for these to System V Make, Andrew Hume's mk, and other unnamed sources.) The major improvements include transitive closure of the dependency graph and implicit intermediate files (rule chaining). An *include* facility and conditional execution based on the C preprocessor language eases the maintenance of makefiles since common definitions and machine dependent parts can be stored separately and dynamically chosen. GNU Make extends the syntax for implicit rules to allow simple patterns.

Dmake. Dmake [52] has many of the same extensions as GNU Make such as transitive closure of the derivation graph (rule chaining), an include mechanism, and better rule definition syntax. Dmake further extends Make by supporting parallel construction of a target's prerequisites. The syntax for targets is extended to force sequential construction when desired. The greatest enhancement that Dmake provides is that it saves the state of a Makefile (e.g. macro expansions, tools used, tool flags, etc.) with each target so it can detect out-of-date targets due to changes to a Makefile.

Imake. Imake [10] is a tool for generating Makefiles. It compiles application dependency information along with architecture descriptions to create Makefiles, which are then interpreted by standard Make. The C preprocessor macro language is used to describe systems. Changes to a description require regenerating the Makefiles. Careful use of directories is needed to avoid inconsistent systems (e.g. mixing sparc and mips object files).

Odin. Odin [14] is a system for integrating existing tools into a single environment. Tools are described declaratively and then linked into a *derivation graph* that summarizes all the "type correct" derivations possible using the current set of tools. Given a request for a particular object, the derivation graph is used to infer build steps. As with Make, the programmer must specify all the dependencies of a configuration. Version selection is not integrated into the system, but Odin supports annotations on targets to perform variant (e.g debug versus optimized) selection. Odin uses a separate directory for each tool invocation and creates a derived object cache that associates the provenance with each object so that requests for a different variant are correctly handled.

6.2 Environments

The systems described in this section support more than building software. They include assistance for describing software and they store meta-data about components in addition to their contents. This meta-data is used for configuration management, version selection, and history management. All of the following systems provide good support for building; they differ primarily in how systems are specified, how version selection is accomplished, and what kinds of consistency are guaranteed.

DSEE. DSEE [30, 31] is a commercial environment that manages software in a network of distributed (Apollo) workstations. It supports a notion of "time travel" by compactly storing versions of source files and providing a tool, the History Manager, that associates symbolic attributes with particular versions. A separate tool, the Release Manager, maintains groups of consistent files. DSEE configurations start with a user-supplied dependency relation called the *system model*. Version selection rules are used to bind object references in the system model to specific versions in the file system. DSEE supports many other features needed in a distributed environment such as transparent access to remote files.

ClearCase. ClearCase [7] is a new product from the original DSEE developers that provides many of the same features on a variety of Unix platforms. ClearCase tightly integrates build management and version selection. As with DSEE, source components are versioned and version selection is accomplished with rules. ClearCase extends the versioning model to include directories and modifies the Unix filesystem to create dynamic views based on the current selection criteria (their filesystem is called a Versioned Object Base). ClearCase decouples building from configuration management, allowing any build tool to be used. They provide a Make utility called ClearMake that does "build auditing" to automatically record exact dependencies during a build by tracking file system *open* calls. Derived objects are shared between different developers automatically. ClearCase supports parallel builds using a network of workstations. The repository can be replicated and distributed; changes made to each site are merged on a periodic basis.

Vesta. The Vesta system [33, 13, 21, 9] developed at DEC SRC is a repository and SCM language that uses a functional model. All source components are immutable. Changes are made using a tool that builds a new configuration object from a previous version by replacing new or modified components. The immutability of components (including configuration objects) means that rebuilding a derived object that was erased is easy and is guaranteed to create the same value as before. Derived objects are treated as accelerators for functional expressions and are cached using an algorithm that generates an almost perfect hash of an arbitrary stream of bytes. The functional system models and component immutability are borrowed from Cedar/SML [29].

CaseWare. CaseWare [12] is a commercial SCM system that manages sources, documentation, and derived information. It uses an object-oriented repository that supports extensible attributes. Software systems consist of component hierarchies and CaseWare has graphical tools to create and modify them. An adaptable process model is tightly integrated with the environment and it provides support for life-cycle management, change management, and problem tracking. The environment performs transparent builds and generates Makefiles on demand to support incremental building. Each object is versioned and rules, similar to DSEE, perform version selection of the components in an system.

Shape. Shape [35, 36] integrates Make with a version control system similar to RCS. Shape is backwards compatible with Make and adds version selection rules comparable to those in DSEE. These rules use regular expressions to specify an ordered list of version preferences, such as "use the newest version of all components I am working on", and "use the newest stable version of all other components." The default selection rule is to use the most recent version of all components as in Make. Shape stores source objects in an attributed file system and can distinguish different versions of objects by using "version attributes." Shape's use of Makefiles is convenient, but such files contain a static description of the system, and maintaining that description becomes more and more difficult as the system grows larger. Like DSEE, Shape relies on an external tool for checking that a configuration is consistent.

Jason. Jason is a *generic* software configuration system [53] that constructs a software environment from a given set of parameters. These parameters include class definitions (object schemas), consistency constraints, and build plans (dependency relations). Created environments can later be extended, but because of the "compiled" nature of generated environments, such extensions are limited to additions and refinements. This limitation prevents Jason from supporting certain kinds of evolution (such as re-organizing a two-component system into a three-component system). Jason uses a powerful constraint language (full first-order predicate calculus) and compiles the given constraints into procedures that check the consistency of configurations. A rigorous algebraic model provides Jason with a strong theoretical foundation not present in most other systems.

Adele/Nomade. Adele [17, 18] (and its successor Nomade [19]) is a constraint-based environment for SCM. An attributed filesystem is used to store information about components. Constraints are quantifier-free boolean expressions whose domain is the attribute values. System architects describe systems using constraints and Adele uses them to infer consistent configurations, including the dependency relation between components. Recent work has extended Adele to include process support based on events and triggers [8].

SMILE/Marvel. SMILE and Marvel [23, 24, 25] are two rule based environments that emphasize support for the edit/debug/build cycle. Their goal is to provide a "fileless" environment for programmers by making the environment responsible for invisibly maintaining derived objects. Their rules use Hoare-style pre and post conditions to trigger actions; they can be used for either forward-chaining or backward-chaining inferences. Forward-chaining corresponds to opportunistic computation and backward-chaining corresponds to the method employed by traditional build tools such as Make. Recent work [26] has extended Marvel to include rule-based process support.

6.3 Logic-based Approaches

The similarity between the rules and dependency statements in a makefile and a logic programming language has been noted by others. Logic and logic-programming languages have been used as description languages and as the basis for build tools like Make. All these systems share the desire to use the power of logic languages for consistency checking and make good use of the richer data model provided by terms.

Feature Logic. The NORA [55] system describes components and configurations using feature logic. Feature logic has as its data model terms which are similar to POL terms (both data models are derived from work by Aït-Kaci [4]). The operations of feature logic are selection, complement, intersection, and union of feature terms. Unification in full feature logic is not as easy as in POL since it includes both negation and disjunction. NORA uses feature logic to represent both revisions (changes made to a component that are intended to replace the existing component) and variants (changes in the behavior of a component), providing a uniform representation for these traditionally separate concepts. A subset of feature logic is mapped into the C preprocessor language and used for their prototype editor. Using this editor, programmers interactively pick values for feature terms. The editor supports incremental selection of features and dynamic sensitivity to existing features. In the future they plan to implement a build tool based on feature logic.

Deductive Databases. Asirelli and Inverardi [6] describe how a deductive database could be used to assist configuration management and software construction. Their system (EDBLOG) is a deductive database consisting of three elements: facts, rules and integrity constraints. They implement a Make-like tool by using facts to represent information about components such as their names and last-modified-dates. Rules are used to define implicit and explicit dependencies and to define the "out of date" relationship. Integrity constraints are used to represent transactions, which correspond to invoking the traditional Make command. Concepts like the "history of a component" are defined by rules and enforced by integrity constraints. Their design only covers building; they did not extend their ideas to configuration or version management.

Prolog-based Make. PROM [27] is a Make tool implemented in Prolog. It defines a small language for writing the equivalent of makefiles. Prolog terms are used to represent information, so the data model is an improvement over Make's character string model. In addition logical variables can be used to implement implicit derivation rules. The initial knowledge base is constructed from the "prom-files" and timestamp information of the files constituting an application.

7 Status and Future Plans

A prototype of CAPITL has been implemented that includes all the basic components: the object-base, worlds, EFS, the Congress interpreter, a browser, a planner and a builder. The planner and builder subsume the functionality of Make. We have tested them on small, but complicated examples that have about 10 source objects and 20 tool descriptions. They have been successfully used to construct CAPITL itself, which consists of about 32,000 lines of C++ code and 5,000 lines of Congress code. The planner correctly generates plans, the builder constructs executable binaries (running tools such as the C compiler with the aid of EFS), and the resulting programs can be invoked through the interactive user interface. After changes to only the *contents* attribute of source objects, the planner quickly verifies the existing expressions and the builder is able to rebuild targets taking advantage of unchanged sources and intermediate objects.

Logic-based construction has many advantages. Complex systems are described compactly and precisely easing maintenance costs. Consistency can be guaranteed and programmers freed from having to worry about how to generate a particular variant of a system. Knowledge about tools is shared so not every programmer must be a tool expert. Types and sub-types are easily represented using POL terms. Abstraction using logical variables and constraints is a powerful method of specifying software.

Several areas need more attention before CAPITL can be successfully used in large-scale development.

7.1 Performance

The planner and builder are currently an order of magnitude slower than Make. Although the incremental algorithms reduce this overhead, it is still a significant problem when a version of a system is initially planned. Experiments with the planner showed that for a given project the order in which tools were considered affected performance either upwards or downwards by over fifty percent. Because most projects are use a small set of tools, simple heuristics such as "try the last successfully used tool" should limit the variation due to fixed tool orderings. An estimate of tool costs would enable the planner to use better heuristics for finding an initial plan such as a branch-and-bound strategy.

The performance of the builder is dominated by the cost of applying tools to components. EFS is considerably slower than native or NFS file system access, particularly for updating files, so tool invocations (such as compiling) are severely degraded. The main source of the problem appears to be inappropriate timeouts and buffering strategies in the NFS client code in the Unix kernel. A successor to Exodus, called Shore [15], is being developed by a separate project; one of the goals of Shore is to provide a facility similar to EFS with higher performance.

7.2 User Interface

Several enhancements to the user interface are needed to make CAPITL usable by system architects and developers. From these higher-level interfaces, CAPITL would generate Congress code, effectively using Congress as an "assembly code" of system descriptions.

Syntactic Sugar. Currently, source objects, tool descriptions, templates and goals are all created "by hand" as Congress expressions. Forms-based interfaces would help to streamline this process and eliminate the need for developers to learn about logic programming. The browser understands a few attributes and has special displays for them (for example, the `contents` attribute of a plain file object is displayed in a text window), but most attributes are simply displayed using the Congress expression language. More special-case displays would help.

Idioms. Common terms such as frequently used forms and functionalities (such as the subterm `Lex_Func` in Figure 4 could be collected in libraries and displayed by the graphical interface as icons for pasting into new descriptions. Such terms could be abstractions (*i.e.* contain free variables) from the project domain. For example, the behavior of a generic translator that preserved functionality might have parameters for the source and target languages and could be represented by the term

```
Translator : function(
   in  => object( form => Source, functionality => Func )
   out => object( form => Target, functionality => Func)).
```

They can also serve as the basis for creating new types through subtyping.

Worlds. Currently, a user must set the default world either by navigating the tree of worlds, or by explicitly typing version ID's as sequences of integers. Worlds could be represented explicitly in the object-base as special "world objects" so that mnemonic names and other attributes can be associated with them. Congress could then be used as a query language for the collection of worlds.

Flexibility. The graphical interface of CAPITL uses different views based on the C++ class of the object being visited. The compiled nature of the interface limits the possibilities for customizing the view of an object based on its dynamic type (*form* and *functionality* attributes). A CAPITL object is self-describing. Descriptions of its representation, behavior, and relationships are contained in its associated attributes. This description could be extended to include a graphical view as well. Interfaces based on embedded interpreters such as TCL/TK [38, 39] or a binding of Congress with X windows intrinsics could be used to implement this extension.

7.3 Additional Functionality

The user interface enhancements suggested above are all fairly straightforward. Other usability enhancements require more fundamental research.

Version Management. Versioned worlds are simple to understand, but policies for managing them need to be explored. CAPITL contains no provisions for mediating access to mutable worlds except that provided by Exodus, which only serializes simple updates; more sophisticated kinds of long-term locks are needed. Mechanisms for selecting worlds and maintaining their internal consistency need to be developed. Tools are needed to support an "algebra" of modifications that allow modifications to be added and subtracted. An example of "addition" is reconciling and merging concurrent updates that created sibling worlds. An example of "subtraction" arises when two updates were applied in sequence, and it is desirable to generate the world that would have arisen if the second update were applied but not the first.

Cache Management. A *derived attribute* is one whose value is an immutable function of other attributes. Derived attributes can be elided or recomputed even in a committed world. Currently, a derived value must be deleted by hand. A useful tool would be one that selectively flushes cached values based on their size, time since last use, and cost of reconstruction. More generally, all derived objects may be thought of as a residing in a cache. In some cases there may be more than one equivalent way to build a product. For example, an executable program might be rebuilt from its sources or reconstituted from a compressed version. Thus cache maintenance is intimately tied to the larger issue of planning.

7.4 Other Possibilities

In addition to software development activities related to the construction of software, there are two areas where CAPITL could help. Both of these areas have the potential to reduce the cost of develop software and increase the reliability of the final product.

Reuse. Considerable attention is now being given to reusing software in order to lower the cost of writing new software [1, 2]. Any reuse system must provide a way of storing and locating components. CAPITL would be a good foundation for such environments because of its rich data model, flexible policies, and extendible attributes. In addition Congress is well suited as an ad-hoc query language for locating components for reuse.

Process Support. Most of the effort in CAPITL has been spent on making the job of the software developer and architect easier. In large projects, the majority of time is spent on managing these developers and on communication costs [40, 54]. Support for such activities is called *process programming* or *process modeling*. The requirements for effective process support is an active research area. Nonetheless, there is agreement that in order to support, as well as control, human activities, flexibility is needed. CAPITL has a good data model and language on which to build process support.

7.5 Additional Functionality

8 Acknowledgments

Many people have helped to bring CAPITL to its current state. S. T. S. Prasad was the first to explore the application of Prolog to configuration management. Tony Rich carried this work forward. His thesis is the basis for many of the ideas in Section 4 of this paper, and the current planner and builder owe much of their content to his work. Odysseas Tsatalos, Theoharis Hadjiioannou, and Trip Lazarus have made substantial contributions to the X-based interactive interface. Lazarus also implemented the *worlds* mechanism and integrated EFS into CAPITL. Delta files were implemented by Tsatalos. Tom Ball and Sam Bates offered many helpful comments on earlier drafts of this paper.

References

1. *Proceedings of the 14th International Conference on Software Engineering.* IEEE Computer Society Press, May 1992.
2. *Proceedings of the 15th International Conference on Software Engineering.* IEEE Computer Society Press, May 1993.
3. Paul Adams and Marvin Solomon. POL: Persistent objects with logic. Technical Report 1158, University of Wisconsin—Madison, Computer Sciences Department, August 1993.
4. Hassan Aït-Kaci. *An Algebraic Semantics Approach to the Effective Resolution of Type Equations*, pages 293–351. Number 45 in Theoretical Computer Science. Elsevier Science Publishers B.V., 1986.

5. Hassan Aït-Kaci and Roger Nasr. LOGIN: a logic programming language with built-in inheritance. *Journal of Logic Programming*, pages 181–215, March 1986.
6. P. Asirelli and P. Inverardi. Enhancing configuration facilities in software development: A logic approach. In *Proceedings of ESEC '87 : 1st European Software Engineering Conference*, volume 289 of *Lecture Notes in Computer Science*, pages 55–63. Springer–Verlag, September 1987.
7. Atria Software, Natick,MA. *ClearCase Concepts Manual*, document number 3000-002-a edition, 1992.
8. N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: A support to large software development process. In M. Downson, editor, *Proceedings of the 1st International Conference on Software Process*, pages 159–170, Redondo Beach, CA, October 1991. IEEE Computer Society Press.
9. Mark R. Brown and John R. Ellis. Bridges: Tools to extend the vesta configuration management system. Technical report, Digital SRC, June 1993.
10. Todd Brunhoff and Jim Fulton. Imake - c preprocessor interface to the make utility. Technical report, Unix Programmer's Manual, X Window System, Version 11, Release 5.
11. Michael Carey, David DeWitt, Goetz Graefe, David Haight, Joel Richardson, Daniel Schuh, Eugene Shekita, and Scott Vandenberg. The EXODUS extensible DBMS project: an overview. In Stan Zdonik and David Maier, editors, *Readings in Object-Oriented Databases*. Morgan-Kaufman, 1990.
12. CaseWare, Inc, Irvine, CA. *Introduction to CaseWare/CM*, part number ic-031-010 edition, 1993.
13. Sheng-Yang Chiu and Roy Levin. The vesta repository: A file system extension for software development. Technical report, Digital SRC, June 1993.
14. Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.
15. David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *SIGMOD*, Minneapolis, MN, May 1994.
16. James R. Driscol, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.
17. J. Estublier. A configuration manager: the ADELE database of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Harwichport, MA, June 1985. GTE Laboratories.
18. J. Estublier, S. Ghoul, and S. Krakowiak. Preliminary experience with a configuration control system. In *Proceedings of the SIGSOFT/SIGPLAN Software Eng. Symposium on Practical Software Development Environments*, pages 149–156. ACM, April 1984.
19. Jacky Estublier. Configuration management: the notion and the tools. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 38–61, Grassau, W. Germany, January 1988. B.G. Teubner, Stuttgart, W. Germany.
20. Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.
21. Christine B. Hanna and Roy Levin. The vesta language for configuration management. Technical report, Digital SRC, June 1993.

22. Stephen C. Johnson. YACC—yet another compiler compiler. C. S. Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.
23. Gail Kaiser and Peter H. Feiler. Granularity issues in a knowledge-based programming environment. In *2nd Kansas Conference on Knowledge-Based Software Development*, Manhattan, KA, October 1986.
24. Gail Kaiser and Peter H. Feiler. SMILE/MARVEL: Two approaches to knowledge-based programming environments. Tech. Report CU-CS-227-86, Department of Computer Science, Columbia University, New York, NY 10027, October 1986.
25. Gail Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 80–88, Monterey, CA, March 1987. IEEE Computer Society Press.
26. Gail E. Kaiser. A rule-based process server component for constructing rule-based development environments. In *Proceedings of the 7th International Software Process Workshop*, pages 76–78, October 1991.
27. Thilo Kielmann. Using prolog for software system maintenance. In *Proceedings of the First International Conference on the Practical Application of Prolog*, volume 1, London, UK, October 1992.
28. Kevin Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.
29. Butler W. Lampson and Eric E. Schmidt. Organizing software in a distributed environment. *ACM SIGPLAN* Notices, 18(6):1–13, June 1983.
30. David B. Leblang and Robert P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. *ACM SIGPLAN* Notices, 19(5):104–112, April 1984.
31. David B. Leblang and Gordon D. McLean, Jr. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122–127, Harwichport, MA, June 1985. GTE Laboratories.
32. M.E. Lesk. Lex—a lexical analyzer generator. C. S. Technical Report 39, Bell Laboratories, Murray Hill, NJ, October 1975.
33. Roy Levin and Paul R. McJones. The vesta approach to precise configuration of large software systems. Technical report, Digital SRC, June 1993.
34. Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, pages 8–24, February 1989.
35. Axel Mahler and Andreas Lampen. Shape—a software configuration management tool. In *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, W. Germany, January 1988. B.G. Teubner, Stuttgart, W. Germany.
36. Axel Mahler and Andreas Lampen. An integrated toolset for engineering software configurations. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, number (24)2 in ACM SIGPLAN *Notices*, pages 191–200, February 1989.
37. John K. Ousterhout. Tcl: an embeddable command language. In *1990 Winter USENIX Conference Proceedings*, 1990.
38. John K. Ousterhout. Tcl: An embeddable command language. *1990 Winter USENIX Conference Proceedings*, 1990.
39. John K. Ousterhout. An x11 toolkit based on the tcl language. *1991 Winter USENIX Conference Proceedings*, 1991.

40. Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta, Jr. Finding out what goes on in a software development organization. Technical report, Personal communication (paper submitted for publication), September 1993.
41. Anthony Rich. *Logic-Based System Modelling*. PhD thesis, University of Wisconsin—Madison, August 1991.
42. Anthony Rich and Marvin Solomon. A logic-based approach to system modelling. In Peter H. Feiler, editor, *Workshop on Software Configuration Management*, pages 84–93, Trondheim, Norway, June 1991.
43. Joel Richardson, Michael Carey, and Daniel Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3), July 1993.
44. Russell Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, OR, June 1985.
45. R.W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 16(8):57–69, August 1983.
46. Marvin Solomon. EFS: the extensible file system. Unpublished manual, 1993.
47. Richard M. Stallman and Roland McGrath. *GNU Make*. Free Software Foundation, Inc., Cambridge, MA, edition 0.26 beta edition, 1990.
48. Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Managing change in software development through process programming. Technical Report 1, University of Colorado at Boulder, June 1991.
49. Walter F. Tichy. RCS: a system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.
50. Walter F. Tichy. Tools for software configuration management. In Jürgen F.H. Winkler, editor, *Proceedings of the First International Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, FRG, January 1988. B.G. Teubner, Stuttgart, W. Germany.
51. Walter F. Tichy and Mark C. Baker. Smart recompilation. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 236–244, New Orleans, Louisiana, January 1985.
52. Dennis Vadura. Dmake—maintain program groups, or interdependent files. Technical report, University of Waterloo, 1992. Dmake Unix Programmers Manual, Version 3.8.
53. Douglas Wiebe. *Generic Software Configuration Management: Theory and Design*. PhD thesis, University of Washington, Seattle, 1990.
54. Edward Yourdon. *The Decline & Fall of the American Programmer*. Prentice Hall, 1993.
55. Andreas Zeller. Configuration management with feature logics. Technical report, Technishe Universität Braunschweig Report No. 94-01, March 1994.