
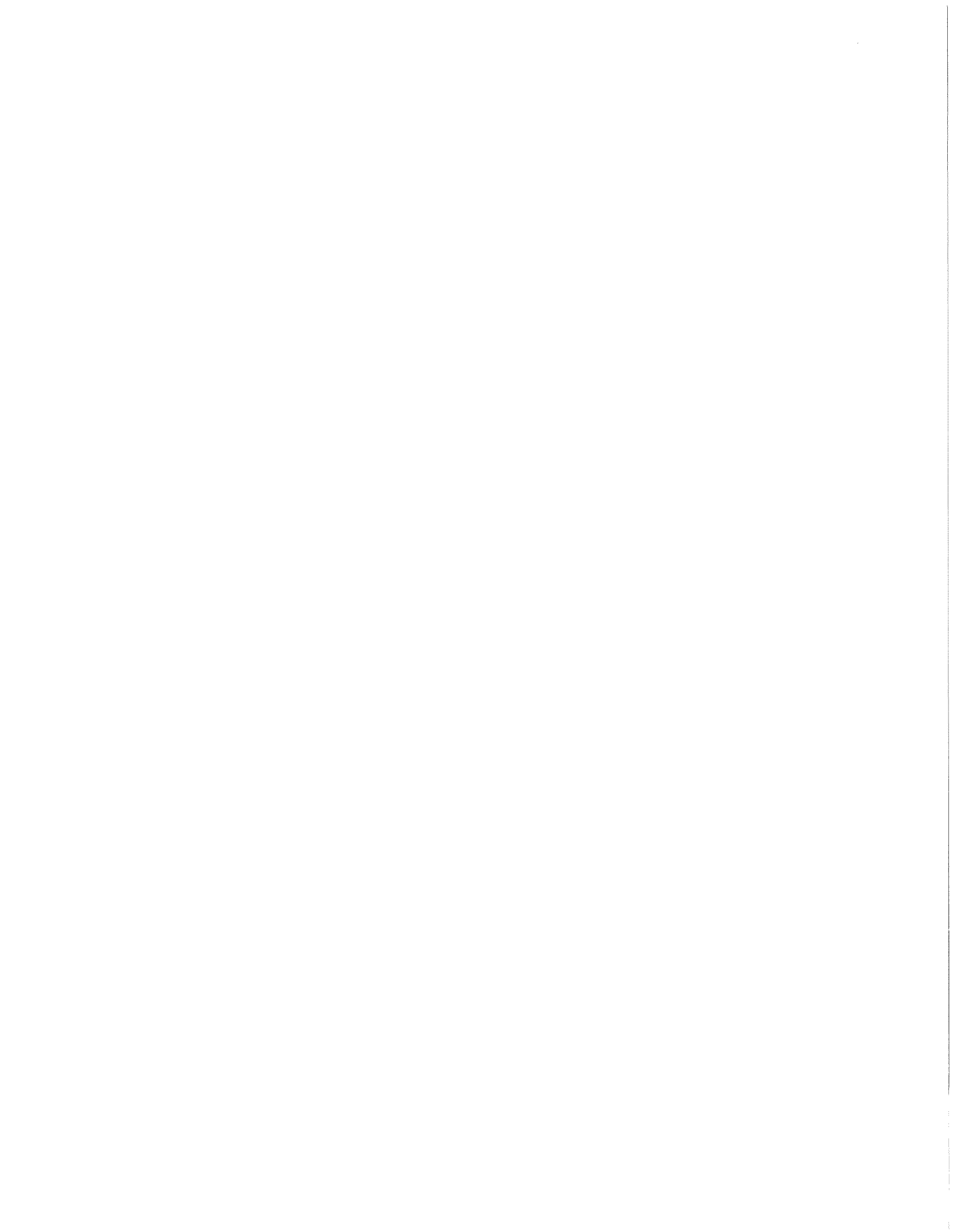


A Logic-Based Approach to System Modelling 

by
Anthony Rich
Marvin Solomon

Computer Sciences Technical Report #928
April 1990



A Logic-Based Approach to System Modelling*

Anthony Rich
Marvin Solomon
Computer Sciences Department
University of Wisconsin—Madison

Managing a complex software system requires a description of how its components are related to one another. We propose a new approach to this problem based on logic programming. All objects in the system are defined by *terms* that describe their format, functionality, components, and origin. Each of these properties is itself a pattern that can be matched against other object descriptions by the unification algorithm. Thus, for example, the *format* of a Pascal compiler written in C is “C”; its *functionality* it is to translate an object whose format is “Pascal” into an object with format “object code” while preserving functionality. The inference algorithm is sufficiently powerful to describe cross-compiling and bootstrapping (that a compiler can compile itself). Our approach is sufficiently flexible that all aspects of the software configuration can be specified in a single language. It allows a complex specification to be split into manageable pieces. For example, the input/output behavior of a compiler can be specified separately from the fact that this behavior is implemented by combining the functionalities of parsing and code generation.

We present the approach in detail by working through an extended example. We discuss its advantages and limitations, and outline our plans for further research on its applicability to a variety of problems in system specification and building.

1. Introduction

The Software Configuration Management Problem

Software configuration management (SCM) systems provide automatic assistance in managing large systems of software objects having complex interrelationships. The software management task is difficult because large systems are continually revised, repaired and reorganized.

Some SCM systems attempt to model a continuously evolving software system by focusing on some aspect of the system that is assumed invariant or at least changing slowly. For example, a system model might describe the system’s structure, based on the assumption that overall structure is revised infrequently. Unfortunately, software systems seem to have the property that nothing is invariant; all aspects of a system are open to rapid change, especially during early development. Other SCM systems deal with this inherent unruliness by creating rule-based or law-based models [Kaiser1986, Minsky1988] which attempt to impose one or more desired invariants on a software system, such as “interfaces must always match,” or “a release can never include an untested component.” A good system model documents the architecture of a software system, guides the process of building derived objects from their antecedents, and specifies consistency constraints among its components. Maintenance of the model should be automated as much as possible.

*This work supported in part by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590.
Authors’ address: 1210 W. Dayton St., Madison, WI 53706, solomon@cs.wisc.edu, (608) 262-1204.

To build a large software product correctly, the proper software components must be selected¹, the proper derivation tools must be applied to those components, and derivations must be done in the proper order. Our approach is to design a logic-based expert system which is capable of automatically satisfying build-request goals. Such an expert build tool requires access to the following items:

- A build-request goal for one or more desired derived objects.
- The objectbase of source modules (and a cache of derived objects, for efficiency).
- Relationships which hold among software objects (e.g., module B is used by module A).
- Descriptions of the capabilities of the derivation tools available.
- A definition of system consistency for the software system under development (e.g., timestamp ordering between source and derived objects, source interfaces match, etc.).

For a given software system under development, these items constitute the system; to the extent that they are not self-describing, they constitute the system model. That is, we consider a system model to be the portions of a given software system which are not self-describing and thus require an external specification. To minimize the system model, we attach extensive descriptive information to each component of the software system and its development environment.

A Logic-Based Approach

In the following examples we show how each of the items above may be defined using a single logic-based language, a variant of Prolog² called LOGIN [Ait-Kaci1986]. A database of persistent software objects is treated as though it were a Prolog-style clausal database so that a build tool written in LOGIN can manage persistent objects directly. LOGIN's inheritance-based unification is used to achieve flexible associative access to objects and to infer the derivation tools needed to process source objects during a build.

The remainder of this paper is organized as follows: In section 2, we describe the LOGIN language and present a small example that illustrates how we can handle simple build tasks like those automated by the popular UnixTM build tool, *make* [Feldman1979]. Section 3 fleshes out the example, showing in more detail how we use terms to represent tools and other software objects. Section 4 describes how queries can be written which invoke a logic-based build and how those queries can be processed to derive atomic or aggregate objects. In section 5 we discuss some of the advantages of our approach. Finally, sections 6 and 7 describe the current status of our work and suggest avenues of further research.

2. A Simple Example

To show how a system model guides the tool inferencing process during a build, we present a simple example comparing our logic-based approach to the procedural approach used by the Unix build program *make*. *Make* uses a procedural system model called a *makefile* to guide a build. Since Unix files are

¹The important issue of version management is beyond the scope of this paper.

²We assume throughout this paper that the reader is familiar with Prolog.

Unix is a trademark of AT&T Bell Laboratories.

normally updated in place, *make* compares timestamps to determine whether a requested file is obsolete with respect to its sources and so must be rebuilt. *Make* uses a limited form of pattern matching to summarize the action of tools³.

```
.y.c:: yacc $*.y ; mv y.tab.c $*.c
.c.o:: cc -c $*.c
.o.x:: ld -o $*.x $*.o
```

The first rule tells how to translate a *yacc* source file [Johnson1975] into a program in the C programming language [Kernighan1988], by applying the *yacc* tool and renaming its output (the notation “\$*” represents the prefix of the file name matching the input pattern). Similarly, the second rule describes the action of the C compiler, and the third describes the Unix linkage editor *ld*.

In our approach, the tools *cc*, *yacc*, and *ld* are self-describing; that is, the tool-specific information expressed in *make*’s type-mapping rules is stored in the representation of each particular tool rather than in the builder program. An important part of every tool’s representation describes its behavior or *functionality*. A tool’s functionality describes both its static interface and its semantics. An interface consists of the tool’s input and output constraints expressed as a pair of partially specified objects. The input-object pattern acts as a precondition or input filter; only an object which unifies with that input pattern will be accepted as an argument to the tool. The output pattern acts as a weak postcondition, describing the output object that the tool can produce. A tool’s semantics are partly described by *coreference constraints*, which express relationships which will hold between the tool’s input and output objects after successful execution of the tool.

Our characterization of tools in terms of patterns acting as pre- and postconditions is somewhat similar to the approach used by Marvel [Kaiser1986]. An important difference is that we use unification to match objects against declarative tool input-output descriptions rather than by procedurally evaluating Boolean predicates. We borrow the concept of viewing tools as type-conversion operators from Odin [Clemm1990].

The representation of the C compiler (with some details suppressed) might look like the following.

```
object (
  name => cc,
  syntax => x,
  semantics => function(
    input => object (
      name => N: ANY,
      syntax => c, semantics => InputSemantics: ANY),
    output => object (
      name => N,
      syntax => o, semantics => InputSemantics)))
```

Descriptions similar to that of *cc* above can be written for *yacc* and *ld*; only the value of the input and output attribute *syntax* would differ in each.

³For simplicity, we use the nonstandard suffix *.x* in this paper to indicate executable programs.

The C compiler is described as having syntax `x`—that is, it is an executable program. Its semantics is described by the `function` term which has two attributes: `input`, which is an object obeying the syntax of the C language, and `output`, which is a relocatable object module. A coreference constraint is defined by the tag `InputSemantics`. Although the value of this attribute is unconstrained (ANY), it appears in both the `input` and `output` terms, which states that although the compiler makes no assumptions about the semantics of its input, that semantics is preserved in the compiled program.

The notation above resembles that of *feature structures* used in natural language processing; in fact, it is an adaptation of the language LOGIN, which was originally developed for expressing constraints in natural language. LOGIN, a variant of Prolog, has two extensions that significantly clarify our notation: Arguments are matched by keyword labels rather than by position, and coreference constraints are indicated by *tags*, which prefix subterms.

Terms can be used not only to describe tools and other existing objects, but also to express requests to build new objects having particular properties. An existing C source program might be described by the term

```
object(name => prog, syntax => c, semantics => numbercrunch).
```

A request for an executable version of this program can be expressed by the term

```
object(name => prog, syntax => x),
```

which asks for an executable program having the name `prog`. Notice that we said nothing about the desired semantics of the result; if a prankster had replaced our number-crunching program `prog` with a semantically different one having the same name, the build request would still succeed. If we are primarily concerned with obtaining the correct semantics in the result, we could guard against such mischief by instead specifying

```
object(syntax => x, semantics => numbercrunch).
```

in which case we obtain an executable program having the desired semantics (but not necessarily produced from or resulting in an object named `prog`).

The LOGIN interpreter works backward from the query to the existing object, noting the sequence of tools which must be applied (much as *make* does). The request is used to search the objectbase for an existing object matching the request. If none is found, the request is unified with the `output` attribute of tool (executable) objects. The output of the tool `ld` unifies with the request; as a side-effect, the `input` attribute of `ld` becomes further instantiated and becomes a new goal to be satisfied. Similarly, this goal unifies with the output pattern of the `cc` term, generating a request for a source object, which unifies with the existing C program `prog`.

3. Object Descriptions

A critical aspect of our design is the description of objects within the objectbase. If a backwards-chaining build tool is to produce a complete and correct build plan for a complex software object, it must have a very precise description of the form and functionality required in the desired result. Thus an object's

representation must include a description of its semantic functionality as well as its external physical attributes. For example, several tools may have identical input-output signatures but perform very different transformations, so a means of expressing the nature of the particular transformations is required. In this section we present object descriptions in detail and show how descriptions of various kinds of objects are combined to describe a complex software system.

An object description is a structure with (at least) six attributes:

Name. A human-readable character-string name for the object, corresponding roughly to the prefix portion of a Unix file name.

OID. An object identifier that uniquely specifies a particular instance. The OID is useful for describing permanent bindings among objects, especially in an object's *provenance* (see below).

Form. A more detailed version of the `syntax` attribute in the simple example above. The form of an object may be atomic, a homogeneous list (i.e., the subobjects all have the same form), or a record (not all subobjects have the same form).

Functionality. A more detailed version of the `semantics` attribute in the simple example above. The functionality of an object might simply be a character-string name (actually, a term with no attributes) if further description of its functionality is not relevant to the build process. The functionality of a tool includes a signature describing its input-output behavior. The tool signature takes the form of two partially specified formal objects describing the essential characteristics of the tool's (possibly composite) input and output. A tool is matched with actual inputs and outputs by unifying these partial specifications with existing objects or signatures of other tools in the objectbase.

Composition. The composition of an atomic object simply consists of its internal contents. For list or record objects the composition consists of (possibly associative) references to its components.

Provenance. A provenance⁴ is a record of how a derived object was created, including references to the tool that created it, the input to that tool, and a creation timestamp. The provenance is useful for historical information and for consistency checking; for example, we may be interested in a version of a derived object that was built from a particular source or using a particular compiler. Provenances can also help maintain a derived-object cache. Before invoking a tool, we can compute the provenance of its output. If an object having that provenance already exists, the tool need not be invoked, thus saving a potentially expensive derivation step. This approach is used in Apollo's *DSEE* system [Leblang1984, Apollo1984], and is more reliable and more flexible than *make*'s technique of using timestamps and *post hoc* reasoning⁵.

⁴This term was suggested to us by an article by Jon Bentley: "The provenance of a museum piece lists the origin or source of the object. Antiques are worth more when they have a provenance (this chair was build in such-and-such, then purchased by so-and-so, etc.). You might think of a provenance as a pedigree for a nonliving object." [Bentley1987]

⁵post hoc, n. \ˈpost-ˈhɑk\ [NL *post hoc, ergo propter hoc* after this, therefore because of this] : the fallacy of arguing from temporal sequence to a causal relation.

The latter four attributes (form, functionality, composition, and provenance) have previously been suggested as descriptive attributes for objects in a more-general context under the names "formal cause," "final cause," "material cause," and "efficient cause," respectively [Aristotle357 B.C.].

An object reference is simply a pattern which matches the object's representation. An object reference can be used as a key for either a direct or associative search by matching it against the objectbase using unification. Specifying an object's OID produces a direct reference to a specific object; leaving the OID value unbound and specifying an object's functionality and form results in an associative search for an object having the specified characteristics.

Example

`c_compiler` is a term which describes the general functionality of a C compiler. A particular C compiler might be represented by the term `cc`.

```
c_compiler := translator(
  input => object(
    form => atom(c),
    functionality => InFunc: ANY),
  output => object(
    form => atom(vax_reloc),
    functionality => InFunc)).

cc := object(
  oid => 51742,
  name => cc_vax,
  form => atom(vax_exe),
  functionality => c_compiler,
  provenance => prov(
    tool => object(oid => 43417),
    arg => object(oid => 40998),
    time => 1989.11.06.10.47.56),
  composition => [ -- compiler's object code is stored here -- ]).
```

The definition of `cc` can be read as follows.

form: The C compiler is a single VAX™ executable module.

functionality: The object is a translator that accepts a C source module as input and produces a relocatable module as output. Although the input and output have distinct forms, they have the same functionality. A cross compiler would be described by changing the `form` sub-attribute of the output attribute.

`provenance`: The C compiler was produced on November 6 1989 at 10:47:56 by applying tool 43417 (probably a version of the linker) to argument 40998 (probably a composite object comprised of a list of relocatable object modules).

`composition`: The composition of the C compiler is its executable object code, which is stored as the value of this attribute.

4. Rules

Rules are used to represent, request, plan, and build the objects within a software system. LOGIN distinguishes between *type constructors* used to build terms and *predicate symbols* used “at the top level” to build assertions among terms. (Prolog makes no such distinction; Prolog “functors” are used for both purposes). We use the predicate symbol `build` to denote a request to find or build an object matching a given term, and `built` to denote that a given object exists. For example, the existence of the C compiler described previously would be stated as the “fact” (rule with empty body)

```
built(cc).
```

A request to build such a C compiler might be expressed as

```
?- build(object(functionality => c_compiler)).
```

A more specific request can be framed simply by specifying more attributes. For example, a C compiler that runs on a VAX is requested by the query

```
?- build(object(functionality => c_compiler, form => vax_exe)).
```

A build request is satisfied immediately if the requested object exists:

```
build(Target: object) :- built(Target: object).
```

More complex build rules describe how one object may be derived from others. For example, the rule

```
build(Target: object(form => atom)) :-
    build(Tool: object(
        form => atom(vax_exe),
        functionality => translator(
            input => Source, output => Target)),
    build(Source: object),
    apply(Tool, Source, Target).                                     (2)
```

states that a target object can be built by applying a tool to a source object, provided the source and tool are built first, the tool is an executable translator, and the source and target match the input and output patterns of the tool’s functionality. The predicate `apply` has the side effect of executing the object bound to `Tool` using `Source` as input and binds the `OID` and `provenance` attributes of `Target`.

Derivation tools (such as link editors) that take lists of inputs require a more elaborate specification.

```
linkedit := translator(
  input => object(
    form => list(atom(vax_reloc)),
    functionality => InFunc: ANY),
  output => object(
    form => atom(vax_exe),
    functionality => InFunc)).
composite_func(PCompiler, [pascal_parser, pascal_codegen]).      (3)
```

```
build(object(form => list(Form), functionality => Func)) :-      (4)
  composite_func(Func, Funcs),
  distribute(Form, Funcs, Components),
  build_list(Components).
```

```
distribute(Form, [], []).
distribute(Form, [Func | Funcs],
  [object(form=>Form, functionality => Func) | Objects]) :-
  distribute(Form, Funcs, Objects).
```

```
build_list([]).
build_list([Head | Tail]) :- build(Head), build_list(Tail).
```

The functionality of the link editor looks superficially like that of the C compiler except that it takes an input of form `list(vax_reloc)` and produces an output of form `vax_exe` while preserving functionality. However, the fact that compiler functionality is obtained by combining a module that parses with one that generates code is a design decision that must be specified manually (rule (3)). Using rule (4), the inference engine can determine how to build an executable compiler if source modules with functionality `pascal_parser` and `pascal_codegen` exist.

5. Advantages of a Logic-based Approach

The examples above illustrate what we believe to be many important advantages to our approach to system modelling. A logic-based approach allows us to employ a single language for representing objects, describing tool signatures, expressing relationships and consistency constraints, defining inference rules, matching objects, and building derived objects. The language is declarative. Programmers specify the properties of tools and other objects; strategies for satisfying requests are inferred. Since the language is declarative, it can be statically typechecked using inheritance-based unification.

A system description can be conveniently decomposed so that each piece of information may be supplied by the person best able to provide it: The author of a primitive object (source module) declares its intended functionality, the designer of a system describes how the functionalities of its components combine, the developer of a tool specifies its capabilities, and the implementor of the SCM environment provides the strategies for deriving objects.

The use of a single language throughout an object management system reduces overall complexity, but at the risk of compromising its expressiveness for a particular purpose. Fortunately, a logic-based language with inheritance provides important capabilities which actually make it convenient for each particular use. For example, using inheritance-based unification as a means of associative access to objects

provides a powerful yet declarative method of object retrieval; it adapts the concept of query-by-example [Zloof1977] to the task of system modelling. Since object types can be expressed as partially specified objects [Wiebe1988], types can be incrementally refined to more-specific subtypes in subsequent definitions. This capability provides one of the often-touted advantages of object-oriented programming: the ability to reuse and build on existing definitions. Finally, the use of a logic-based language makes it easy to create what is essentially an expert system for the construction of complex software objects from a collection of primitive components.

6. Current Status

Development of these ideas is underway as part of the the CAPITL⁶ project [Solomon1990]. We have implemented a dialect of LOGIN called *Congress*. (LOGIN was first implemented in Prolog at MCC, but it is not publicly available; we used C++ [Stroustrup1986] as our implementation language.) Congress also incorporates some ideas from a more recent extension of LOGIN called LIFE [Ait-Kaci1988, Ait-Kaci1988a]. The examples in this paper have been coded and tested. We have built and tested a prototype that extends the techniques presented in this paper.

The example above uses a “build as you plan” strategy for creating complex objects. Tools are applied as a side effect of searching for a derivation path from existing sources to the requested object. If no such path exists, unnecessary work will be done. A better approach responds to a request for an object by building a *plan* to create the object. If this planning step succeeds, the resulting plan can be executed to create the object.

The plan is not simply a functional expression. The existence of a derivation path does not guarantee the success of a build; individual tool applications may fail, as when a compiler is applied to a program containing errors. However, alternate paths may exist which could succeed, so the plan itself must be prepared to backtrack. Thus, rather than writing a *builder* as described in this paper, we have implemented a *planner* that emits a logic program as its output. Plans can be quite complex. For example, in some software systems, executable versions of derivation tools must be built and executed dynamically as part of the build itself. A plan can be viewed as a version of the configuration description specialized to build a particular target.

We are also implementing an object database using the Exodus extensible database toolkit [Carey1988]. The runtime support for Congress will present existing database objects as terms.

7. Directions for Further Research

Our initial experiences with a logic-based language for SCM are very encouraging. However, more research is needed to establish this approach as a practical tool for defining all aspects of a complex software system.

⁶Computer Assisted Programming-In-The-Large

More Examples. We need to apply the technique to a wide variety of existing software systems exhibiting complex configurations with different combinations of languages, tools, and interactions. We believe we can handle derivation steps other than compilation/linking, such as interpretation, document production, program integration [Reps1988], file format conversion, and testing, but we will not know what additional mechanisms are required to support these applications until we actually try to model them.

Version Management. Effective integration of version management with the build process requires development of a representation for version families and version histories. Mechanisms are also required to support selection among multiple functionally equivalent objects according to criteria such as "most recent", "marked stable", or "tested against a standard test suite". Research in this area is ongoing [Solomon1990].

Multiple Provenances. Support for automatic integration [Reps1988] requires multiple provenances. For example, the same object may be derived in different ways by integrating several objects in different orders; a derived object may therefore have several different but equivalent provenances. This situation also arises whenever a reversible operation (e.g., data compression) is applied to an object. In general, the different derivations which can produce a given object will have different processing costs. When a requested derived object has been removed from the derived object cache and must be rederived, the provenance selected to rebuild the object should be the one whose total rederivation cost is lowest.

8. References.

Ait-Kaci1986.

Hassan Ait-Kaci and Roger Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance," *J. Logic Programming* 3(3) pp. 187-215 (March 1986).

Ait-Kaci1988.

Hassan Ait-Kaci and Patrick Lincoln, "LIFE: A Natural Language for Natural Language," MCC Tech. Report ACA-ST-074-88, MCC, ACA Program, Systems Technology Laboratory, 3500 West Balcones Center Drive, Austin, TX 78759 (February 1988).

Ait-Kaci1988a.

Hassan Ait-Kaci and Patrick Lincoln, "LIFE's Rich Tapestry: A User Manual and Reference Guide to the LIFE Programming Language," MCC Tech. Report ACA-ST-408-88, MCC, ACA Program, Systems Technology Laboratory, 3500 West Balcones Center Drive, Austin, TX 78759 (February 1988).

Apollo,1984.

Apollo Computer Inc., *DOMAIN Software Engineering Environment (DSEE) Reference*, Apollo Computer Inc., 330 Billerica Road, Chelmsford, MA 01824 (1984).

Aristotle 357 B.C..

Aristotle, *Metaphysics*. ca. 357 BC.

Bentley1987.

Jon Bentley, "Programming Pearls: Self-describing Data," *Communications of the ACM* 30(6) pp. 479-483 (June 1987).

Carey1988.

M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview," Computer Science Tech. Report #808, University of Wisconsin-Madison, Madison, Wisconsin, 53706 (November, 1988).

- Clemm1990.
Geoffrey M. Clemm and Leon J. Osterweil, "A Mechanism for Environment Integration," *ACM Transactions on Programming Languages and Systems* 12(1) pp. 1-25 (January 1990).
- Clocksint1984.
W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin Heidelberg New York Tokyo (1984).
- Feldman1979.
S. I. Feldman, "Make—A Program for Maintaining Computer Programs," *Software—Practice and Experience* 9(4) pp. 255-265 (April 1979).
- Johnson1975.
Stephen C. Johnson, "YACC—Yet Another Compiler Compiler," C. S. Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ (1975).
- Kaiser1986.
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).
- Kernighan1988.
Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1988).
- Leblang1984.
David B. Leblang and Robert P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices* 19(5) pp. 104-112 (April 1984).
- Minsky1988.
Naftaly H. Minsky and David Rozenshtein, "A Software Development Environment for Law-Governed Systems," *ACM SIGSOFT Software Engineering Notes* 13(5) pp. 65-75 (November 1988).
- Reps1988.
Thomas Reps and Susan Horwitz, "Support for Integrating Program Variants in an Environment for Programming in the Large," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 197-216 B.G. Teubner, Stuttgart, W. Germany, (January 27-29, 1988).
- Solomon1990.
Marvin Solomon, *An overview of the CAPITL programming environment*, In preparation. 1990.
- Stroustrup1986.
Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA (1986).
- Wiebe1988.
Douglas Wiebe, "Partial Instantiation: A Data Model for Versioning Software Databases," Tech. Report 88-04-04, Department of Computer Science, University of Washington, Seattle, WA 98195 (April 1988).
- Zloof1977.
M. M. Zloof, "Query-by-Example: A Data Base Language," *IBM Systems Journal* 16(4) pp. 324-343 (1977).

