(11)

Beyond the Read-Eval Loop: The Artifacts System

Mike Karr

February 14, 1994

1980 3960 458

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

The purpose of the Artifacts System is to structure complex, evolving data, to assist users in their cooperative effort to develop such data, and to integrate the tools that operate on and produce this data. A key element in the design is to eliminate what is the usual interaction with a computer-based system: run a tool to achieve a desired effect. Rather, users of the Artifacts System set up structures that indicate desired results and browse these structures in hypertext-like fashion; tool invocation is usually implicit. Version and configuration management is an integral part of the system, not a facility on the side.

Keywords: configuration management, tool integration, version control

This work was supported in part with funds provided by ARPA, under contracts N00014-85-0710 and N00014-90-0024.

Contents

1 Introduction

The Artifacts System supports the development and evolution of products—such as software, documents, eletrical or mechanical designs—where the following characteristics are typical:

- The data is large and highly structured.
- The data evolves, usually linearly, but occasionally with branches and merges.
- Several people work jointly on a project.
- The derived information is expensive to compute and to store.
- The set of tools that compute derived information changes (usually grows).
- There is a heterogeneous network of computers that can be exploited as a whole.

These factors invite a change in what has been the usual paradigm for interacting with a computer-based system: run a tool to achieve a desired effect. Whether the system is Lisp-style, where the interaction is a successive evaluation of forms, or UNIX-style, with its successive execution of programs, the end result is that the user spends time trying to manage a vast sea of relatively unstructured project data. In the Artifacts System, a user sets up a structure that specifies a desired result, and the system is responsible for the details of tool invocation.

The Artifacts System is both a framework—whose particulars are primarily of interest to extenders when incorporating a tool—and an environment—whose particulars are of interest to end users. (We have no dogma regarding the choice of these words "framework" and "environment"; we only mean to convey a distinction.) To an extender, the Artifacts System offers a small functional interface for adding tools; a simple way to wrap, rather than rewrite, existing tools; extensive support for highly incremental tools; and support for fine-grained tools. These features affect how the end user views the Artifacts System as well, from whose perspective the system provides

- a uniform interface (to a collection of tools);
- automatic configuration management and version control;
- automatic tool invocation if desired;
- scheduling of concurrent tool invocation across a network;
- reliable rederivation of results produced from prior versions;
- support for literate programming; and
- hypertext-like browsing.

All but the last of these say something about tools. This is because a goal of the Artifacts System is, in fact, to enable the end user to forget about the mechanics of tool invocation and focus instead on the contents of the product. The last item supports the end user's typical activity when using the Artifacts System—that of editing. As a point of emphasis, configuration management is a consequence of the way the system works, not an additional activity for the user.

The description of the Artifacts System that follows exposes both perspectives—the extender's and the end user's—as does the system. In fact, a further goal of the Artifacts System is to offer an open framework that is easy to extend. We mention the distinction merely in case the reader finds it a helpful when we describe the underpinnings, or system's view.

2 Fundamental Relations

An *artifact* is the fundamental unit of data in the system. Like a file, an artifact has contents used to store data of interest to users and, like a file system, the Artifacts System does not interpret the contents. An artifact differs from a file in that *its contents never change*. In this respect, it is more like a "version" of a file, as long as one's notion of a file system does not allow changing the contents of a version of a file. The term "artifact" was chosen with its archaeological connotation in mind: small, created some time ago, and unchanging. The immutability property may seem surprising at first, but, as we shall see, it plays a key role in providing a proper basis for tracking the evolution of a large and constantly changing structure, such as a program or document.

An artifact has a type, and typing of artifacts plays a central role in integrating tools. The set of types grows as new tools are added to the environment. Classical file systems often use naming conventions to achieve something of the effect of types, but they generally do not enforce a correspondence between a naming convention and the contents of a file. The Artifacts System uses types of artifacts in a classical programming way: to govern the structure of the contents of an artifact and the operations that can be applied to it. The contents of an artifact may be ordinary text or more structured, a topic we discuss more fully in section 4. Extenders of the environment provide new types whose operations conform to prescribed rules (analogous to "method interfaces" or "roles" in object-oriented programming).

Both file systems and the Artifacts System have "fine structure" and "coarse structure" of data. We will use the term "fine structure" to refer to the bytes in files or to the contents of artifacts, which are typically more structured. Indeed, the precise nature of the contents is not under the purview of the system. The term "coarse structure" refers to the constructs that the system provides and uses in locating and grouping the fundamental units of data. In a file system, the coarse structure is the directory structure, by which we mean the conceptual structure exposed by its interface. For example, the coarse structure of the UNIX file system has soft and hard links, as well as other constructs detectable by UNIX system calls. The following fundamental relations are the "coarse structure" of the Artifacts System:

- references—connecting fine and coarse structure;

- derivative—recording the results of tool invocation; and
- successor/predecessor—to track the evolution of artifacts.

(The following sections discuss these relations in more detail.)

Another property of file systems is that the names by which a user locates files are an integral part of the coarse structure; they typically name branches in the directory. The same names are used by programs that access files. In the Artifacts System, programs use a notion of an artifact "pointer", a unique identifier, to access artifacts. These pointers are involved in an open-ended set of attributes or other structures, a topic explored more fully in section 6. The "name" by which a user locates an artifact is just one of the attributes of an artifact.

2.1 References

A *reference* from one artifact to another occurs at a particular point in the contents of the referencing artifact. For example, the artifact that specifies this document refers, immediately below, to an artifact of type `C-module`:

The `printline` module

——————— *Targets* ———————

——————— *Source* ———————

[stdio.h]⟨spec⟩

```
global ⊥   void
printline(s)
  char *s;
{
  printf(s);
  printf("\n");
}
```

What appears in this document is the typeset form of the artifact; in the interactive use of the system, references appear as highlighted regions on the screen. Commands that take artifacts as arguments default to an artifact near the cursor, if possible. Since one of these commands is "examine-artifact", the overall effect is that of browsing through hypertext.

Most of the rest of this section develops this example into a complete C program to illustrate several points about the Artifacts System:

- the use of references to indicate dependencies;
- the use of a structure (constructed by following the references relation) to specify implicitly the invocation of tools to produce a desired result; and
- the tight integration of code and documentation, using many references to small artifacts. (The document itself is the illustration.)

The first two points address the goal to reduce the bookkeeping burden on the end user, and the last addresses the goal to support literate programming. At the end of the section, we introduce an important definition.

The first line of the above inset is the caption. We postpone discussing the targets section until we get to the next example of a `C-module` artifact, which has a non-empty target section. The purpose of the source section is to supply both text to be compiled by a C compiler, corresponding to the usual `.c` file, as well as the interface, or *spec*, that other modules need to use the public functions defined within it, corresponding to the usual `.h` file. The first line of the source section corresponds to `#include <stdio.h>` in an ordinary `.c` file. It is the typeset form of an artifact reference to an artifact of type `universal`, an artifact type we discuss in section 6.

The user of the Artifacts System has the option of defining the spec (`.h` file) for a `C-module` artifact implicitly. In the above module, the definition of the `printline` function is marked as "global" in the artifact, and the spec for this module is constructed to contain the suitable declaration for `printline`. (The typeset version of the artifact marks such regions with a marginal tag.) The advantage of using this style is that the header of a function appears in one place, so it is impossible to have an inconsistent declaration and definition.

The following artifact refers to the spec of the previous one:

The `hello` module

——————— *Targets* ———————

sun4 [gcc (universal artifact)] **-g**

——————— *Source* ———————

```
  [The printline module]⟨spec⟩

  void
main()
{
  [The body of the main program for hello]
}
```

This example has a non-empty targets section. In general, a targets section in a `C-module` artifact may have 0 or more lines, each of which specifies a particular compilation of the module. In this case, the one target line specifies the machine architecture "sun4" and that the compilation will be done with gcc, again using a `universal` artifact to link into the file system; the rest of the line (-g) specifies compiler switches.

We have already discussed the first line of the source section. The body of the function `main` has a reference to an artifact without ⟨spec⟩. This is to a `C-source` artifact, the purpose of which is merely to allow the text that the compiler will see to be broken into small pieces and scattered through a document, as we do here:

The body of the main program for `hello`

```
printline("Hello world!");
```

This particular example is excessively fine-grained and intended only as an illustration, both

of references and of the way in which the Artifacts System supports the tight integration of programs and documentation. (See the end of section 2.3 for more about LaTeX.)

To complete this example, the following `Unix-program` artifact (deliberately uncaptioned) specifies an executable for each of two architectures.

```
————————— Targets ——————————————————————————————
sun4-dbg  [cc (universal artifact)] -g
  C [cc (universal artifact)] -g
mips-pro  [gcc (universal artifact)]
  C [gcc (universal artifact)] -O
————————— Modules/Instructions ————————————————
  [The hello module]
```

The targets section of this artifact has two subsections (each beginning with a non-indented line), one for the sun4 architecture and one for the mips. The respective variants (dbg and pro) are simply to name the switch combinations (for the linker) that follow. (Non-blank variants are not necessary here, but would be if there were more than one target for a given architecture.) The dbg variant supplies the -g switch, and the pro variant, no switches. On the first line of each subsection is an artifact reference to the linker. In the usual C style, we have used the same program for linking that is used for compiling, but this is not required; for example, one could use `ld` directly.

The indented lines of a target subsection specify how to compile the modules of the program. Each such line begins with a language $\mathcal{L}$ (here, "C") and has an artifact for a compiler, followed by switches. The machinery has much more flexibility than needs to be explained in this paper, and we give only the rules sufficient to explain the above examples.

- If the targets section of a module artifact has an entry for the same architecture and variant as that for the executable target, use the object module specified by the module artifact.
- Otherwise, if the module is written in language $\mathcal{L}$ and if there is a line in the subsection of the `Unix-program` artifact beginning with $\mathcal{L}$, use the compiler and switches on that line to compile the module.
- Otherwise, issue an error.

Thus, even though the `C-module` artifacts did not say how they should be compiled for the mips, it is still possible to use them in a mips executable. Also, the compilation specified for the `hello` module is not used in the executable on the sun4, because there, the variant is blank, while the `Unix-program` artifact wanted a `dbg` variant. (Features not discussed here allow one variant of a program to use differing variants of modules.)

The Modules/Instructions section of the `Unix-program` artifact provides the modules that make up the executable, together with instructions (switches) to the linker (for the case in which switches need to be between modules—here, there are no switches). Note that only the `hello` module is given and not the `printline` module. The default rule is that modules on which other modules depend are included in the list of modules given to the linker. Thus, if a programmer uses artifact references to indicate dependencies, the system guarantees that all the relevant modules are put into the executable. As with the automatic construction of `.h` files, the idea is to reduce the bookkeeping burden on the programmer.

We mentioned earlier that an artifact is somewhat like a version of a file. In this section, we have seen that artifacts have references to other artifacts. This is a bit like a file embedding the names of other files, or rather a version of a file embedding the names and versions of other files. However, there is a crucial difference: unlike a file system (or most configuration management systems, such as RCS [Tic85]), *the Artifacts System knows which artifacts refer to other artifacts* at the coarse level. (The locations of the references within the fine structure of an artifact are known only to corresponding type-specific tools.) This property means that there is a natural formal definition in the Artifacts System for a term that is widely used intuitively:

- A *configuration* is the artifact structure in the transitive closure of the references relation starting at a given artifact (the "root").

The Artifacts System does not allow the deletion of an artifact referenced from another artifact, thereby guaranteeing the integrity of configurations. Note also that because of multiple references to an artifact, the number of artifacts in two configurations may be far less than the sum of the numbers in each.

When we integrated language tools into the Artifacts System, we chose to represent the compilers explicitly as artifacts. Thus they become literally part of a configuration.

2.2 Derivatives

In section 1, we stated that "a user sets up a structure that specifies a desired result". We can now be precise about exactly what that structure is: a configuration. This section concerns the "desired result", which we call a *derivative*. In order for the statement to hold, it is necessary that a derivative be completely determined by the configuration rooted at the artifact. If we view the derivative as being associated with the root of a configuration, the rule is that a derivative of an artifact depends only on the contents of that artifact and on referenced artifacts, recursively. The rule that a derivative cannot depend on information outside artifacts is why, for example, compilers and switches appear in artifacts—they are responsible for producing a derivative. The rule is also intimately linked to the immutability property of artifacts: the derivative associated with a configuration is always the correct one for that configuration. If someone changes something, that something is a new artifact, not the one with which the derivative is associated. The immutability rule thus simplifies both the user's view and the implementation of a configuration management scheme in a highly distributed multiuser world.

An artifact may have several *kinds* of derivatives. For example, the `Unix-program` artifact above has derivatives of kinds `executable%sun4-dbg` and `executable%mips-pro`. Similarly, the `hello C-module` artifact has an `object%sun4` derivative. Technically, the derivative relation consists of triples $\langle p, k, d \rangle$, where p is the artifact pointer (unique identifier), k is the kind, and d is the derivative; at most one such triple exists with any given p, k pair. Derivatives are stored as the k-derivative of an artifact. The effect of this in the user's view is that derivatives never rattle around loose in the Artifacts System, in contrast with file systems in which, for example, locating the files that go into a particular executable is by itself an uncertain process, never mind trying to determine what procedure and what options might have been used to derive the `.o` files incorporated into the executable.

The user usually deals directly with artifacts, not their derivatives. For example, the command to execute a program takes an artifact and a host as arguments and looks for a derivative whose kind is of the form **executable%***architecture-variant*, where *architecture* is the architecture of the given host. If there is precisely one *variant* for the machine, then that derivative is used; if more than one, the user is asked to specify the *variant*. Thus, the user need not be aware of derivatives as such and can deal instead with notions having to do with artifacts.

A *deriver* is a tool that takes a single artifact as an argument and produces derivatives. The requirement that a deriver take a single argument is not a real limitation: all it says is that what one ordinarily thinks of as arguments must be gathered into a single artifact. This requirement supports the simple model for tracking derivatives described above and has other advantages that we describe later. Far from being limiting, the idea that an artifact defines the inputs for a deriver is the basis for the openness of the Artifacts System: it is extended by the joint additions of derivers and types. For example, the purpose of the **Unix-program** type is to integrate linkers. Similarly, the purpose of the $\mathcal{L}$**-module** type is to integrate compilers for the language $\mathcal{L}$.

A key element in the design of derivers is that they work by scanning the contents of an artifact and requesting particular kinds of derivatives of references, regardless of the type of the reference. For example, a **Unix-program** artifact is not restricted to referring to $\mathcal{L}$**-module** artifacts; rather, the requirement is that the module referred to must have a **module-summary** derivative, which in turn records what language the module is written in, which **object%...** derivatives it supplies, and the transitive closure of dependent modules. Thus, the kind says what is wanted from an artifact, and the type says what its structure is: the two together specify a somewhat object-oriented approach to locating a deriver, similar to that found in other tool-integration systems, e.g., [Har87].

The principle that a deriver look only at the derivatives of references is important in the ease of extension of the system. By narrowing the scope of derivers, it makes writing them easier. The system takes care of exploiting network-wide computing resources (similar to [LJ87]). This all works in the following manner:

- Phase 1—The deriver scans the contents of the artifacts to determine which kinds of derivatives are needed from which references. It structures this as a list of pairs $\langle p_i, k_i \rangle_i$, which we call a *request*, and submits it.

- Interphase—The system accepts the request, figures out which of the derivatives already exist, and, if necessary, schedules jobs to compute the rest. If a job to produce the derivative is already scheduled or running, a new job is not scheduled. The jobs are run concurrently as resources become available, and eventually all the derivatives $\langle d_i \rangle_i$ exist; the system sends these to the deriver.

- Phase 2 (optional)—Using the derivatives obtained so far, the deriver submits new requests to gather more derivatives, iterating until sufficient information has been obtained. (This involves an interphase for each additional request.)

- Phase 3—The deriver uses the collected derivatives to construct the derivative(s) for this artifact and informs the system (so that it can record it and give it to jobs that may be waiting).

The difficult part of the implementation is the "interphase" part, but extenders don't have to worry about this—they just provide a serial program and the system worries about the concurrent scheduling. In addition to providing concurrent tool invocation, the Artifacts System accommodates the heterogeneity of the network. Information as to which tools can run on which hosts is supplied to it when installing tools and hosts. It uses this information to exploit the network-wide resources.

2.3 Successors and Predecessors

The successor relation is a simple binary relation on artifacts; the predecessor relation is its inverse. They are used to track the evolutionary relationship of artifacts. The number of successors and predecessors is unconstrained, so this can be used to record branching and merging.

In addition to its obvious role in history keeping, the successor relation also plays a role in detecting simultaneous edits, as follows. An artifact is defined to be *out-of-date* if it has a successor or if it references an out-of-date artifact. (Thus the root of a configuration is up-to-date if and only if all the artifacts in the configuration are up-to-date.) When a user asks to supersede an out-of-date artifact, i.e., establish a new successor for it, the system tells the user that the artifact is out-of-date and asks if the user wants to proceed. If the user goes ahead, branching will occur—but the user has been warned that a departure from linear evolution will occur.

Unlike the contents and references of an artifact, the sets of successors and predecessors are not immutable. We have just seen that new successors can be added, and both successors and predecessors disappear when an artifact is deleted. In general, deletion and these relations interact in the following way: if A is a predecessor of B and B is a predecessor of C, then deleting B causes A to become a predecessor of C. Thus deletion changes history, but not violently—it is as if C were originally created as the successor to A.

From a pragmatic point of view, the most important aspect of the historical relations is their role in incremental tools. Because the Artifacts System uses both tool invocation information (specifically, references and types) and historical information (specifically, predecessors), it can be more effective than conventional tools, such as those based on `make` and RCS, that use only naming conventions and timestamps.

- Even though a derivative is *determined* by its contents and references, it may be *computed* by chasing successor/predecessor links and looking at the contents and derivatives of other artifacts.

For example, suppose we add a comment to the above `C-module` artifact. The `C-module` deriver does not scan an artifact and its predecessor to see if only comments have changed, so the module will be recompiled. However, it does check to see if the result of compilation has produced the same object file; since the comment does not affect the object file, it will see these are indeed the same. It then discards the new file and uses the old one as the derivative *even though a timestamp would indicate that it is invalid.* The `Unix-program` deriver, which checks to see whether either the contents of the `Unix-program` artifact have changed (modulo references) or at least one of the object modules is different, will see no differences. It can thus tell, without even calling the linker, that the executable will be the same as before and install the old executable as the derivative of the new artifact. The advantages of recording historical and derivative relations explicitly, as opposed to the more common naming conventions and timestamps, should be clear—less chance for error and more opportunity to avoid expensive rederivations. Further, timestamps are not reliable when compilation is done on a network with clocks that are insufficiently synchronized.

Among the derivers installed so far, the one for LaTeX artifacts has the most sophisticated incremental behavior. The user sets up a configuration rooted at an artifact of type `latex-root`. The deriver appropriately runs LaTeX, BIBTEX, and `makeindex` (as well as any typesetting tools for other types of referenced artifacts, such as a `C-module`) to yield a `latex-directory` derivative. The user does not interact directly with the derivative but instead uses commands, such as `preview`, `ps-preview`, `hardcopy`, and `dvi-file` (to export a dvi file), that access the derivative. A `latex-directory` derivative is actually a UNIX directory with the usual files that one has for LaTeX, plus a few more we create for bookkeeping. The `latex-directory` deriver for a `latex-root` artifact looks for the same derivative of a predecessor to determine, for multipart documents, the parts of the document on which LaTeX must be re-run. In many cases, it can avoid runs that a person cannot, because of its bookkeeping files. Part of this sophistication stems from a subversion of the usual LaTeX commands for counters and labels to record whether these entities are referenced and/or set by a particular part; that way the deriver knows what parts are affected by a change. Suppose, for example, that a configuration rooted at a `latex-root` is superseded by one in which the only textual change is to a bibliography entry. The deriver runs BIBTEX; if this causes a change to the appearance of the bibliography, it re-runs LaTeX on the part containing the bibliography; if the change to the entry caused a change to a citation, it re-runs LaTeX only on sections in which the citation appears. Only those who have used the bibliography mechanism of LaTeX can fully appreciate the benefits in hassle reduction and increased reliability that this provides.

3 Editing

Unlike tool invocation and configuration management, editing is not an avoidable activity—it is the means by which users develop their "products", such as programs or hardware specifications. We have designed the system around an editor interface, rather than a shell-like or read-eval interface. However, the Artifacts System is not tied to a particular editor; it is open to the integration of different editors—such as text editors, graphical editors, and syntax-directed editors—just as it is open to the integration of different kinds of tools. Recall that the Artifacts System is insensitive to the fine structure of artifacts; this is the domain of particular editors and tools. An editor does have a responsibility to present references within the fine structure in a natural (and preferably uniform) way and to support commands on, and navigation among, artifacts. Thus the integration of an editor requires more effort than the integration of a tool, which is usually "wrapped" without modification (see section 5). To date, the only editor we have integrated is EMACS, but this has more to do with limited resources than with intrinsic properties of the Artifacts System.

When several people are working together on a single program (or document or circuit design or ...), the problem of making changes to it arises. Some of the difficulties are unavoidable—people may want the data they are developing to incorporate some of the latest changes, but no one wants the latest bugs that are introduced in the process of making those changes. The Artifacts System is not magic, but it does allow several people to work on a program at once, in a reasonably controlled way.

An artifact that is prepared by editing begins its life as a *draft*, which for present purposes is simply an edit buffer whose contents are destined to become the contents of an artifact. Like artifacts, drafts have types. The essential difference between a draft and an artifact is that the contents of a draft change as editing proceeds. A draft may reference (in the technical sense of the word) both artifacts and drafts, but artifacts may not reference drafts—artifacts are stable, and drafts are not. A draft may in fact reference drafts that others are editing.

Because artifacts are immutable, "editing an artifact" actually creates a new artifact. This happens in stages. First, a new draft is created and its contents are initialized to those of the existing artifact; the draft will have a single predecessor, the artifact. One may also start editing from nothing, in which case the draft is initialized in the canonical way for its type, and the predecessor set is null. Once editing is under way, one may "merge" any number of artifacts into the draft, in which case the contents of each artifact are added to those of the draft and each artifact is added to the set of predecessors. (A draft cannot have drafts as predecessors; there seems to be no use for it.)

Regardless of how a draft is started, it is turned into an artifact by *committing* it. This causes the draft to disappear and a new artifact to appear, having the same contents as the draft. If the draft references other drafts, the commit command either first commits the referenced drafts[1] or balks, depending upon the particular command the user issued and whether the referenced draft is the user's own or someone else's (one user cannot commit another's edit). The predecessors of the new artifact are those of the draft. The immutability of artifacts obviates the need for check-in/check-out.

The predecessor relationship for drafts is used not only to determine the predecessor relationship for the eventual artifact, it also plays a role in detecting a simultaneous edit. Analogous to the way that we define "out-of-date", we say that an artifact is *almost-out-of-date* if there is a draft that has it as a predecessor or if it references an artifact that is almost-out-of-date. When beginning to edit or merge an artifact, the user is warned if the artifact is almost-out-of-date. Thus, the branch in lineage is caught when starting a draft, not when committing it.

[1]There is presently no facility for committing artifacts with circular references. This would have to occur in a single atomic transaction, but there are no particular difficulties in doing so.

When a new artifact becomes a successor of an old one, not only does the old one become out-of-date, every configuration of which it is a member becomes out-of-date. It would clearly be unsatisfactory to require the user to edit manually each of these artifacts to change only the references, and then only certain references, to their successors. What happens instead is that starting a draft not only starts the draft the user intended, but also guarantees that certain other drafts exist, automatically creating them if necessary. The default rule is that a draft will exist for every artifact that is up-to-date and will become almost-out-of-date because of the new draft. (Non-default behavior is also accommodated.) A draft that is automatically created is called a *system* draft; such a draft does not correspond to any buffer and differs from its predecessor artifacts only in the references to new drafts. A system draft may be "taken over" for further editing if desired. When committing a draft, the use has the option to "commit upward", i.e., to commit drafts that reference it, recursively, thus bringing into existence successor artifacts for all the artifacts made out-of-date by the commit. One might worry that propagating the effects of editing would consume tremendous amounts of storage. However, as we will see in section 4, the storage required for two artifacts differing only in references is about what it is for one of them alone. Treating them as two artifacts simplifies the bookkeeping of derivatives and provides a simple mental model for tracking evolution.

An important consequence of the fact that the derivative of an artifact depends only upon the configuration rooted at that artifact is that *derivers may be scheduled automatically*. This is the default when a draft is committed and may be overridden if desired. Each job is scheduled and then run when its turn comes and there is a machine on the network capable of doing it. The automatic generation of system drafts, the command to commit upward, and the automatic scheduling of derivers combine to make it easy to create successors of an artifact and all the configurations of which it is part and to obtain derivatives for all the new artifacts—without ever explicitly invoking a single tool. One simply starts the desired draft, makes the modifications, and commits it "upward". With properly incremental tools, the amount of work done is proportional to the effects of the change.

While we have emphasized the implicit use of derivatives and their automatic generation, it is also true that derivatives may be deleted and rederived under user control. Such capabilities are necessary so that a user can control the CPU and disk utilization. Consequently, a user may issue a command that depends upon a derivative that can, but currently doesn't, exist. In such cases, the system asks the user whether it should schedule the job to obtain the derivative, after which the user can re-issue the command.

4 Bodies and Handles

In this section, we discuss a simple techniqu‹ ᶠor the solution of two problems. The first problem, noted in section 2, is the need to ᴗonnect the fine structure of program pieces (recorded in artifacts) with the coarse structure necessary to organize the pieces (the references relation). In addressing that issue, we will sketch how we leave the representation of the fine structure up to the definer of an artifact type (and out of the purview of the system) and how tools use the references relation to wander over structures larger than an artifact.

The second issue, noted in section 3, is the necessity to represent economically two artifacts that differ only in their references.

From the user's point of view, an artifact has contents whose structure is determined by the type of the artifact and has occasional references to other artifacts. From the system's point of view, life is more complicated. In section 2 we mentioned that the system uses an artifact pointer to access artifacts.

- The system associates with each artifact pointer a pair consisting of a pointer to the contents, called a *body*, and a list of artifact pointers corresponding to the references.
- Each reference in the contents of an artifact can be thought of as a consecutive number, called a *handle*. (We say "can be thought of" because the representation of a reference is type-specific and may be more complicated than just an index, as is the case in the example we give in section 5.) The contents of a body never contain an artifact pointer.

(See figure 1.) Tools that deal with artifacts first see an artifact pointer. If thev need the contents of an artifact, they retrieve its body and list of artifact pointers. If they encounter a handle while processing the body, they refer to the pointer list to find the pointer for the referenced artifact. Only type-specific tools know about the structure of bodies, so only such tools can associate handles with artifact pointers. Notice that the system knows nothing about the representation of handles—it never even sees any. A tool is nearly as ignorant of artifact pointers—it uses them only to retrieve a body and pointer list. A serendipitous aspect of recording the list of artifact pointers with each artifact is that this is exactly the references relation discussed earlier.

As all this suggests, an artifact has a "repository form", which is on the disk and which tools see, and a "display form", which is what the user sees via an editor. Handles occur only in the repository form; in display form, they are replaced by references to specific artifacts.

The two-level storage of artifacts is a space-saving device; we described it in detail, in part, to allay concerns about excessive storage consumption (mentioned in section 3). It is also the basis for an efficient, and type-independent, update algorithm: given an artifact A_0 and a pair A_1 and A_1' of artifacts, it produces an artifact A_0' from A_0, where all references to A_1 in A_0 have been replaced by references to A_1' in A_0'. This function works for artifacts of any type, because it never looks inside the body of the artifact, which is the only place that type-dependent structure exists. Put differently, the update function works at the coarse level.

It is quite common for a subset of users, perhaps only one, to want to split off a piece of a system and work on it independently, neither affecting nor being affected by other development work. In file-based environments, this must be done by literally copying files, with non-trivial expense. In the Artifacts System, one also copies artifacts, but the cost is negligible—there is a small overhead per artifact, independent of the size of the contents. Any further cost that accrues is proportional to the amount of actual change that is done. Like the update function, which it resembles, the generic copy function works at the coarse level.

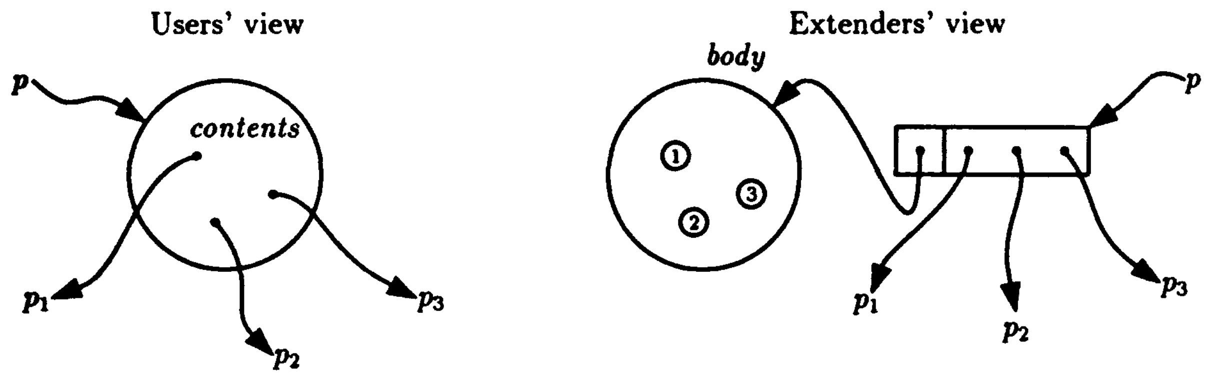

Figure 1: Users' and Extenders' Views of an Artifact p

In the current system, a body is usually a file name relative to a directory specified by the type, but this is only because most of our initial tools have been file-based. (Messing with this part of the file system can destroy the integrity of the artifacts data. This hole must eventually be closed, although it has been an advantage in the development process.) With the advent of persistent objectbases integrated with object-oriented languages, we hope that the day will come when derivers can be written with the same mental model that the user has—contents with references to other artifacts; we expect that the body/handle connection could be encapsulated by the methods of a suitable object class. It is not necessary to make an all-or-nothing decision here. Because the body is interpreted on a type-by-type basis, we can have a system in which some types have bodies that are file names and various other types have bodies that are references to objects in various objectbases.

5 Legacy Issues

The Artifacts System was conceived with objectbase-like underpinnings in mind. The typed artifacts correspond to typed objects and the artifact references, to object references. In such a regime, editors and tools could be written in object-oriented languages without explicitly dropping into text. While this may be an ideal way to write new tools and editors, it is also necessary to accommodate old file-based tools. In this section, we will describe in some detail how we use the type-dependent—some would even say "ad hoc"—representations of fine structure, including references, in order to accommodate wrapping legacy file-based tools.

Consider the standard implementation of the C language, which has no notion of an artifact reference, but whose C preprocessor supports the `#include...` construct. Using such a construct might seem antithetical to the Artifact System's notion of immutability, because the effect of `#include xyz` is dependent upon the contents of the mutable file `xyz`. In fact, we represent a handle in `C-source` and `C-module` artifacts as a `#include` statement, although not in a completely obvious way. What we do is represent a handle i by a macro—

that the C preprocessor will expand—of the form _AS_u_i,[2] where u is the unique identifier of the file in which these contents are found. We then arrange for the compiler to see a small "init" file that defines each of the handle macros to be a reference to the file specified by the appropriate derivative of the appropriate artifact and then `#includes` the "real" file to be compiled, itself stored as a derivative.

References in `C-source` and `C-module` artifacts that are followed by ⟨spec⟩ are somewhat more complicated—the above `#include` construct is embedded in a conditional that insures that `C-spec` derivatives are read only once, a technique familiar to C programmers using `.h` files. The Artifacts System has an advantage over the usual file system convention in that the conditional occurs *outside* the references rather than within the referenced file. When a duplicated reference does occur, this saves opening the file only to skip over the entire contents.

The net effect of all this machinery is that the Artifacts System, with no duplication of data, uses an unmodified C preprocessor and compiler, yet presents the user with an object-oriented, even hypertext-like, environment. The integration of standard C tools uses handle representations that are appropriate for only those tools, exploiting what it can of the capabilities that C offers. The same ideas apply to the integration of other tool suites.

A second legacy issue is that many systems have large libraries of relatively stable information. For example, C has `.h` files in `/user/include` and its subdirectories, and LaTeX comes with a library of "style" files. On the one hand, we do not want to allow arbitrary access of the file system by artifacts; because of the mutability of those files, this would breach the configuration management that the Artifacts System provides. On the other hand, it is unreasonable to copy all of the files simply to make them immutable, when in fact they are quite stable. The compromise is the artifact type `universal`, so called because it allows one to specify an arbitrary set of derivatives. In particular, a `universal` artifact may reference a file (representing a derivative) that is outside the Artifacts System's storage area. Integrity is maintained by storing within the artifact a checksum and modification date. The nightly off-hours integrity check can thus detect if the file changes, up to deliberate duplication of the modification date and 2^{-32} probability of a duplicate checksum. Thus the Artifact System provides a consistent view of immutable artifacts, yet has a pragmatic connection to files and file-based tools.

6 Plexes

Up to this point in the discussion, we have been concentrating on artifacts, whose contents are static. It is clear that underlying the implementation of artifacts must be some mechanism that allows for objects that change. Although the basic ideas in the Artifacts System are independent of the technology used for objects with dynamic contents, the usability of artifacts makes certain demands on the underlying mechanism. This section describes the underlying mechanism both on its own terms and in connection with the Artifacts System.

[2] In time-honored tradition, a symbol beginning with an unlikely prefix (here _AS_ for Artifacts System) is assumed to be an identifier reserved by that system.

From the user's perspective, this is the mechanism that provides "active views". Also, in this section we describe the various attributes of interest to users that we alluded to in section 2.

We use the term *plex* to describe an object whose contents are dynamic. The notion generalizes that of "relation" as it is used in database terminology. A relation is a plex whose state at any one time can be described by a set of tuples, all of the same length. Plexes permit the state to take other forms, for example a list or a tree. (It is understood that relations are logically sufficient; the reason for plexes is that sets of tuples may not be natural and convenient for some purposes.) The essence of plexes is not the data structure nor the operations that change state, but rather their ability to be viewed. Secondly, the Plexes System is extensible: a new plex is defined merely by supplying a set of functions having to do with viewing the plex. We will see that the Artifacts System is an extension of the Plexes System.

A user views a plex with an editor and, to the extent allowed by the plex, changes it with an editor. Because plexes may be large, viewing is controlled by a *filter*, which limits the amount of the plex that the user sees on the screen. While a filter controls what a user sees, a *format* controls how the information is presented on the screen. The details of what a user sees when viewing a plex through a particular filter, arranged according to a particular format, are of course dependent upon that plex, as are the details of how to specify a filter or a format for the plex. However, the notions of filter and format apply when viewing any plex.

For example, the **`artifacts`** plex keeps track of all the artifacts in the system and various attributes attached to them. This includes both information that is fundamental—type, body, references, successors/predecessors, derivatives—as well as information that is for the user—name, the person who created it, time of creation (this is the present set, but it is extensible). For performance reasons, the up-to-date attribute is represented directly, even though it can be computed from the successors and references information.

The **`artifacts`** plex is the means by which a person locates a particular artifact; for example, users generally organize their data so that there is no more than one element in the set of artifacts that is up-to-date and has a particular name and type, or name and owner. The interface for the Plexes System makes locating artifacts in this way no more difficult than specifying a file name. It is also common to use the viewing machinery to locate artifacts in other ways. For example, a user may wish to see all artifacts with a given type that are up-to-date and created before a given time, or all artifacts with a given name, regardless of whether they are up-to-date or what their type is. Such subsets of artifacts may be viewed by specifying the appropriate filter. Similarly, a format may indicate to sort first by name or by time of creation, or may say to include or omit the display of certain attributes.

An important feature of the Plexes System is that views on plexes are "active". Filters provide the ability to control how much information is displayed. Suppose that the user is viewing the **`artifacts`** plex through a filter that allows all users' artifacts provided they are up-to-date and a format that sorts by time of creation; then if any user commits an edit, the lines displaying the now out-of-date artifacts will vanish and a line displaying the new up-to-date artifact will appear in the proper place in the display. This provides the user suffering from file-system withdrawal the illusion of looking at a directory listing that is continuously updated.

To take an example quite different from the **artifacts** plex, let us consider the plexes used in running jobs: **future**, **present**, and **past**. The **future** plex has jobs waiting for available resources. The **present** plex has the current set of resources, specifically, one entry per "job server", the process on each host responsible for running jobs there. The entry will indicate that the job server is idle or have information about the current job. The **past** plex contains a record of all jobs run. A view on these plexes will show a job's progress as it is scheduled, run, and completed.

When we designed the Plexes System several years ago, we imagined that the idea of maintaining active views on a repository was novel. In fact, there were already instances, and the idea has apparently had multiple independent discoveries (such as, Mercury [Lis88], FSD [WA87], and Chiron [TJ93]).

7 Present Status and Future Plans

The Artifacts and Plexes Systems were conceived with a graphics-based window system in mind. The prototype implementation uses Epoch, which in turn is built on GNU EMACS and X, which brings us part, but by no means all, of the way toward our ultimate vision for a bit-mapped user interface. The communication mechanism is based on TCP. The core of the system is written in C, which together with Epoch/EMACS, X, and TCP provides a highly portable, freely distributed base. The whole Artifacts System is public domain.

We currently have around 20 types, for a variety of derivers: LaTeX (used to produce this paper); the C and UNIX types used as examples earlier; Common Lisp and (soon) Ada, in much the same style as C; and shellscripts, which allow integration of UNIX files-to-files programs as derivers.

The Artifacts System is in daily use at Software Options, Inc., and has been distributed to several other sites. It has been sufficiently robust to host its own continuing development for several years. We have done some work to couple the Artifacts System with the Activity Coordination System [Kar93], so that management issues like bug-report tracking and system release policies have a direct tie-in with the development of a project's activities and data.

8 Conclusion

The Artifacts System provides its users with hypertext-like browsing and editing of complex and evolving objects. These objects may be documents, software, hardware or mechanical designs, ..., or like this paper, may combine objects of different kinds. Version/configuration management is an integral part of the system. Tools are so well integrated that there is minimal explicit tool invocation by the user. The coarse structure of the system provides structural relations that connect an object with its historical relatives and with derivative objects, subsuming the roles of naming conventions and timestamp comparisons. As well as providing information of direct interest to users, the coarse structure provides a basis for writing highly incremental tools. The system supports the collaborative work of multiple users on a heterogeneous network that is treated as a single multifaceted resource, with the

Plexes System providing active views on network-wide project data. While the capabilities of the Artifacts System are both powerful and flexible, it has a simple basis: the reference, derivative, and successor/predecessor relations.

9 Bibliography

[Har87] W. Harrison. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4(6):46–56, 1987.

[Kar93] Michael Karr. The activity coordination system. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, September 1993.

[Lis88] Barbara Liskov. Communication in the mercury system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, January 1988.

[LJ87] D. B. Leblang and R. P. Chase Jr. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, 1987.

[Tic85] W. F. Tichy. RCS—a system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[TJ93] Richard N. Taylor and Gregory Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In *The First Collected Arcadia Papers*, pages 47–54, 1993.

[Towa] Judy G. Townley. *E-L Users' Manual.* Software Options, Inc., 22 Hilliard Street, Cambridge MA 02138.

[Towb] Judy G. Townley. *Language Users' Manual.* Software Options, Inc., 22 Hilliard Street, Cambridge MA 02138.

[WA87] David G. Wile and Dennis G. Allard. Worlds: An organizing structure for object-bases. *SIGPlan Notices*, 22(1), January 1987.