A Distributable APSE

S. Tucker Taft
Intermetrics, Inc.
733 Concord Ave.
Cambridge, MA 02138

for: The First International Symposium on Ada for the NASA
Space Station, June 2-6 1986
Nassau Bay Hilton Hotel
Houston, TX

1. Introduction

A distributed Ada(r) Program Support Environment (APSE) is one in which programmers, managers, customers, testers, etc., may work on separate computers, linked by a high-speed network. It also may imply that program development proceeds in a series of relatively independent subsystems, which are then combined into larger Ada programs as part of final integration. (This reminds one of the frequent similarity between the structure of programs and the structure of the organizations that build them.)

This paper will discuss an approach to the implementation of a distributed APSE which provides for parallel development on separate computers while sharing "catalogs" of compiled units, but avoiding global locking or naming bottlenecks.

2. The Ada Program Library

Ada as a language is somewhat unusual in that a "program library" must be maintained across separate compilations, holding compiler- produced information necessary not only for later linking, but also for later compilations. To support a distributed APSE, it is essential that the Ada program library may itself be "distributed," because it is too expensive in disk space and/or compile-time to maintain on each computer a copy of the entire program library.

Even on a single computer, there are reasons to "distribute" the Ada program library. As defined in the Ada Reference Manual (ARM 10.4) the program library holds the "universe" of compilation units available for "WITH" references at compile time, and for eventual linking into an Ada program. Conceptually at least, the library includes all the language-defined packages, such as TEXT_IO, CALENDAR, etc. These by themselves represent a major investment in compile-time and disk space, and most Ada compilation systems have devised some way to share such compiled packages across program libraries.

2.1 Program Library as Network of Catalogs

As a generalization of sharing language-defined compiled packages, we have defined a conceptual Ada program library as a set of interconnected "catalogs," some of which may be connected into other program libraries as well. Each catalog holds a set of (compiled) Ada compilation units represented in a DIANA form, as well as a more conventional object-module form. A conceptual library is constructed from a read/write "primary" catalog plus links to a set of read only "resource" catalogs.

Every program library must provide the language-defined packages, which in our case are

gathered together to form the "RTS" (run-time system) resource catalog. A typical large program might have a series of other resource catalogs for utilities, like a DBMS catalog, MATH catalog, a DEBUG catalog, etc., plus one catalog for each major subsystem.

Each resource catalog is actually part of a set of revisions. Two revisions may share some of their compiled units, and differ in others. We therefore provide for both sharing of compiled units across different program libraries, as well as across revisions of the "same" conceptual program library.

3. The HIF

To support this distributed program library structure in a host independent way, we have defined a standard Host Interface (HIF) to a (distributed) database system. The Hif database is organized as a set of "nodes", partitioned by "Hif user" (where a Hif user maps to a user or sub-project on the Host system). There is a "top-level node" associed with each Hif user, analogous to the "home directory" of a conventional file system.

Hif nodes have string-valued attributes, and relationships from one node to another. The relationships are uni-directional, meaning that they can be viewed as directed arcs in a graph of nodes. A subset of the relationships, called the "primary" relationships, form a strict tree reaching every (non top-level) node by exactly one path. The "secondary" relationships form an arbitrary graph.

3.1 HIF Node Kinds and Partitions

Two kinds of HIF nodes exist: structural and file. File nodes have a host file associated with them (typically containing the DIANA or OBJMOD representation of an Ada compilation unit), while structural nodes serve only as connectors between other nodes, and as carriers of attributes.

The subtree of nodes beneath the top-level node associated with each HIF user, plus all of the host files associated with these nodes form a partition of the HIF database. The information necessary to represent a user's partition is gathered into a single host directory. The node-structure database is represented by 3 files: a B-tree of nodes, a hash-table of relation/key/attribute identifiers, and a heap of attribute values. The file-node host files are assigned HIF-generated names in the host directory.

3.2 Program Library Implementation via the Hif

The program library is implemented using Hif nodes, taking advantage of the partitioning by Hif user. The set of revisions of a resource catalog, plus all of the compiled units included in one or more of the revisions, are combined into a single Hif partition.

In addition, some number of primary catalogs may coexist in the same Hif partition. In particular, the primary catalog used to create the next revision of the resource catalog must be in this same partition.

It is possible to put more than one resource catalog revision set in a single Hif partition. However, maximum flexibility of distribution results from defining a separate Hif user for each resource. Separate partitions for testing help further, by keeping the resource partitions free of test stubs and drivers.

4. Unique Identifiers

Given an Ada program library distributed among primary and resource catalogs, and a Hif database distributed among partitions, a number of interesting technical problems arise in the area of unique naming.

Unique identifiers are needed for compilation unit revisions to correctly determine when a compilation unit goes out-of-date. The compiler must record the unique identifier of all compilation unit revisions referenced while compiling the unit (e.g. the "WITH"ed specs), and then when these are replaced in the (conceptual) program library, the unit must appear out-of-date.

Unique identifiers are also needed for subprograms, so that references at calls may be resolved to the appropriate body. Overloading means a simple string will not suffice.

Finally, unique identifiers are needed for each Ada type, so that strong type checking and overload analysis may be performed correctly. Long identifiers and potentially deep nesting make the full Ada name an inappropriate choice.

In each case it is desirable that the unique identifier be relatively short (e.g. 32 or 64 bits) since there are a very large number of references, and yet be distinguishable from all other identifiers in the distributed program library. This is made more difficult when compiling is proceeding independently on separate computers, presuming there is no centralized assigner of globally unique identifiers.

4.1 Context-dependent Unique Identifiers

We have solved each of these unique identifier problems by using the concept of context-dependent identifiers, with context-dependent translation performed as part of moving the identifier from one context to the next.

4.2 Node Ids, Partition Ids, and Partition Maps

To uniquely identify compilation unit revisions in the distributed Ada program library, we rely on the general Hif node identifier, which consists of two integers, a "partition" id, and a node id. The partition id is simply an index into a "partition map," selecting an entry which identifies the location of the host files representing the partition within the host file system, as well as which partition map (if different from this one) to use for interpreting partition-ids appearing within that partition. The node-id is used as a key into the B-tree (host) file which represents the partition, and is assigned sequentially within the partition as nodes are created.

Each computer can maintain its own partition map relatively independently, assigning its own partition ids. When a reference is created to a partition on another computer that is not yet in the partition map, a partition-id is assigned for use from the referencing computer. The entry in the partition map indicates the location of the partition, as well as the location of the partition map to be used to interpret its partition references. When a node reference (partition-id, node-id pair) is copied from a partition on one computer to a partition on the other computer, the partition-id is translated according to the correspondence between the partition maps on the two computers.

4.2.1 Exporting Partitions and Partition Maps The partition map mechanism makes for a convenient method for exporting a set of partitions on tape, by simply including the partition map on the tape. Then, when the partitions are read back in off the tape, so is the partition map. The partitions are entered into the "master" partition map on the receiving computer, and their entry in the partition map indicates that when interpreting partition references within them, to use the partition map also copied from tape.

For convenience, a partition does not embed its own partition id in self-references, but rather uses the special partition-id zero. This way, if the partition is totally self-contained, there is no need to ship the partition map when shipping the partition all by itself.

4.3 Unique Ada-Entity Identifiers

A second kind of unique identifier, an Ada-entity identifier, must specify a particular Diana node, which represents the entity, among all of the Diana nodes in all of the compilation units in the (distributed) program library. Nevertheless, since there are many thousands of such references in a large program, the node identifiers ("locators") must be kept as small as possible (e.g. 32 bits). This apparently conflicting set of requirements was resolved by making each Diana file its own context for interpreting the locators.

4.3.1 Diana Node Locators; Segment + Offset Node locators are broken up into two halves, 16-bits of segment index, and 16-bits of segment offset. When the segment index is positive, it is an intra-file reference, and the segment index simply selects in which 64K segment of the file the Diana node appears. The segment offset always gives the byte offset within segment.

When the segment index is negative, it as an inter-file reference, and the absolute value of the segment index selects the element in the Diana file's "external segment definition table" which identifies (with a Hif relationship) the compilation unit being referenced, and the segment within it.

This mechanism allows each compilation unit to refer to 32K other compilation unit segments, each of which is up to 64K bytes in length. However, it means that a locator must always be interpreted relative to the file in which it resides. To simplify the manipulation of locators by the compiler, a "master" segment definition table is defined, and all locators are translated to "master" locators as they are retrieved from a Diana file. By design, the master segment definition table becomes the external segment definition table for the Diana file being created at that time, meaning that no additional locator translation need be done on storing into the file being created.

5. Summary and Experience

A distributed Ada program library is a key element in a distributed APSE. To implement this successfully, the program library "universe" as defined by the Ada Reference Manual must be broken up into independently manageable pieces. This in turn requires the support of a distributed database system, as well as a mechanism for uniquely identifying compilation units, linkable subprograms, and Ada types in a decentralized way, to av falling victim to the bottlenecks of a global database and/or global unique-identifier manager.

We have found the ability to decentralize Ada program library activity a major advantage in the management of large Ada programs (in particular, the multi-targeted/multi-hosted Ada compiler itself). We currently have 18 resource-catalog revision sets, each in its own Hif partition, plus 18 partitions for testing each of these, plus 11 partitions for the top-level

compiler/linker/program-library-manager components. Compiling and other development work can proceed in parallel in each of these partitions, without suffering the performance bottlenecks of global locks or global unique-identifier generation.