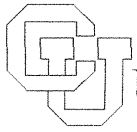
Object Management in a Distributed Software Environment *

Geoffrey M. Clemm
Stanley M. Sutton, Jr.
Lloyd G. Williams

CU-CS-333-86



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Supported in part by the U. S. Department of Energy under contracts no. DE-AC02-80ER10718 and DE-FG02-84ER13283, and in part by the National Science Foundation under grants no. MCS80-00017 and DCR-8403341.

Object Management in a Distributed Software Environment

Geoffrey M. Clemm*, Stanley M. Sutton, Jr.*, and Lloyd G. Williams

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-333-86 June 1986

*Supported in part by the U.S. Department of Energy under contracts no. DE-AC02-80ER10718 and DE-FG02-84ER13283, and in part by the National Science Foundation under grants no. MCS80-00017 and DCR-8403341.

Abstract

This report describes object management in the Keystone project. Keystone is an experimental prototype for a software environment in which software development activities are distributed over a (possibly heterogeneous) network of personal workstations. Its purpose is to explore requirements and design issues for distributed software environments. Keystone is based on a single-machine object manager, the Odin software environment. Odin manages objects local to individual workstations. Management of objects which are imported from or exported to other workstations is accomplished using importer and exporter processes running on each node together with lists of data which describe these objects. The report presents details of this architecture. To date, two levels of prototypes for Keystone have been implemented. Experience in implementing and using these prototypes is also described with particular attention to issues of object management, extensibility, object typing and persistent objects, and support for heterogeneous resources.

Table of Contents

1. Introduction

A software system is a complex entity which is actually made up of a number of individual objects. These objects are the various artifacts of the software process which include specification and design documents, source code, object modules, test cases, and others. One very important function of an automated software environment is the management of these objects.

Management of the objects associated with a software development project is a difficult task. The individual objects are typically quite complex and may themselves be composed from other, also complex, objects. In addition, any given object is likely to be related to many other objects by a number of different relations. Typical relations among objects in a software environment include:

> *Derives:* An object is produced by carrying out one or more transformations on another object. For example, object code is derived from source code via the compiler. In this case, the compiler is viewed as a transformer.
>
> *Constrains:* The state of an object is constrained to bear a particular relation to the state of one or more other objects. An example of the constrains relation would be the maintenance of consistency between corresponding versions of two objects.
>
> *Hierarchical relations:* An object is composed of other objects. The sense of hierarchical as used here is intended to convey more than the typical tree-structured hierarchy. Hierarchical relationships are diverse and can be quite complex, possibly even cyclic. Possible hierarchical relationships include: *imports_from, exports_to, inherits_from, is_an_instance_of, is_an_interface_of, is_a_version_of,* and *is_a_view_of* [Zdon85].

The relationships among software objects (and therefore the management of those objects) becomes considerably more complex if the software environment is distributed (either logically or physically) over multiple hosts. In a distributed environment, multiple copies of objects can exist on different nodes. This introduces additional complexity due to the need to manage concurrent updates and maintain consistency among the various copies in a distributed context. If hardware and/or software resources differ from host to host, the type hierarchy is complicated with additional elements or notations. If, for example, the hosts can have different CPUs, object modules derived from a single source are no longer necessarily equivalent.

In this report, we describe our experience with a simple approach to distributed object management in Keystone, a prototype distributed software environment. Keystone was constructed to explore issues which arise when the various functions of a software environment are physically distributed over a network of workstations. The logical view presented to the user by Keystone is also a distributed one, although facilities are provided to remove much of the burden of dealing with the distributed aspects of the environment.

Three fundamental assumptions embodied in the Keystone approach are that each developer will have his or her own individual workstation, that these workstations will not necessarily have the same CPU, and that each workstation will have its own local

mass storage. (While the use of specialized "file servers" is not precluded, it is felt that reliance on local storage will make the environment more robust and potentially more widely applicable.) The principal design goals for Keystone are:

(1) To develop a distributed environment which is easily extensible by the user.

(2) To provide support for arbitrary, user specified sharing of resources within the environment.

(3) To provide a vehicle to study cooperation and user work patterns in a distributed software environment.

(4) To provide a vehicle for studying the effect of a distributed environment on the software process.

The first two goals support the latter two by making it possible to easily extend or reconfigure the environment to respond to new needs or explore new possibilities.

Authority for object management in a distributed software environment may be assigned in several different ways. First, a single, global object manager may be used. All authority for object management, both for the local node and for distributed transactions rests with the global manager. Second, it is also possible to make use of local object managers from individual nodes if facilities are provided for handling objects in the distributed system. Here, authority for object management is local and local managers cooperate to provide management of distributed transactions. Finally, a "distribution manager" may be used to control object managers on individual nodes. In this case, authority for management of local objects is local while authority for management of distributed transactions rests with the "distribution manager." The first of these corresponds to centralization of authority for object management. The latter two feature decentralized object management with different degrees of coupling between the local object managers.

Keystone uses the second of these approaches. An existing local object manager is augmented with capabilities for importing and exporting objects in cooperation with remote object managers. To date, two prototype versions of Keystone have been implemented on a network containing several different types of processors using version 4.2 of Berkeley Unix.* The configuration of the network which supports Keystone is shown in Figure 1. This report describes the current status of the Keystone project, the architecture of the Keystone object management system, and our experience with its implementation and use.

2. The Object Management System

The overall approach used in constructing the Keystone environment is described in [Clem85]. Briefly, Keystone is based on the Odin software environment [Clem84]. Odin provides an integrated interface to software tools and enables users to extend the capabilities of the environment by adding tools of their own.

The distributed nature of the environment is supported using the Federation concept [Heim85]. This approach, originally developed for distributed database systems,

*Unix is a registered trademark of AT&T Laboratories

supports the sharing of data among autonomous elements (databases) which do not necessary share a common schema. In a Federated system, cooperation replaces central authority and procedures are provided to facilitate sharing. Each member of the Federation specifies the information that it is willing to export and the information that it wishes to import from other members.

Management of objects in Keystone is divided into two distinct spheres of responsibility. Odin manages objects local to individual workstations. Management of objects which are imported from or exported to other workstations is accomplished using importer and exporter processes running on each node together with lists of data which describe these imported and exported objects. In this section we describe the Odin environment from an object-management point of view, extensions to Odin for operation in a distributed environment, and management of import and export operations within Keystone.

2.1. Odin

The Odin system is an automated object manager for single-machine software environments. Objects in Odin correspond to artifacts of the software development process (source code, object modules, test data, etc.). They are, in fact, files in the host file system and the terms "file" and "object" will be used somewhat interchangeably in this report. These objects may be either *simple* or *compound.* Simple objects correspond directly to host system files while compound objects are sets of files. For example, the set of files that makes up the source code for a program is a compound object; the files corresponding to individual modules are simple objects. Odin provides for user-defined object types and various types of objects are distinguished by file extensions, such as ".c" (for a C source file) or ".txt" (for an ordinary text file).

2.1.1. Object Management in Odin

The principal relationship among objects supported by Odin is the *derives* relation. Derivations are controlled using an internal directed graph, called a *derivation graph,* which specifies how the various object types known to Odin are related. An example derivation graph is shown in Figure 2. Each node in the derivation graph represents an object type and each edge represents the derivation required to produce the object at its head from the object at its tail. Tools are thus considered to be transformers which convert objects (files) of one type into objects of another type. For example, the C compiler converts objects of type *C_source* to objects of type *object_code.*

The results of derivations are maintained automatically in a special database and are available for satisfying subsequent requests. This database, together with the derivation graph and command scripts used for tool invocation, resides in a special directory on each node, known (naturally) as the Odin directory. The location of this directory within the overall directory structure is not fixed and may vary from machine to machine or even from user to user on a given machine. It must, however, be known to the system. This directory is also used by some of the Keystone processes and functions, as indicated below.

The derivation graph may be extended by the user to encompass new object types and derivations. This is accomplished by describing object types and their associated operations using a built-in specification language. For example, an object of type

formatted_C_source might be specified as follows:

```
fmt "Formatted C source code"
        "pp <$(c) >$(fmt)"
                :c
```

In this example, "fmt" is a type designation for an object corresponding to formatted C source code. A file containing an object of this type will have the extension ".fmt." The string in quotes on the first line provides a descriptive comment. The second line is the host system command which invokes the formatter (a "pretty-printer"); "$(c)" and "$(fmt) are replaced with the appropriate host system file names at run-time. On the final line, "c" defines the type of object from which a ".fmt" object may be derived.

Odin objects are strongly typed. The set of operations (i.e. transformations) which are valid for a given object are determined from the derivation graph, based on the type of the object. If an invalid operation is attempted (e.g. executing a symbol table), the Odin system will generate an error message and will not perform the operation.

2.1.2. User Interface

Objects are requested by naming the desired derivation. This type of request is known as a *display* request since, when the object is requested, it is displayed to the user in an appropriate fashion. For example, the command

```
test.c
```

would display (i.e. list) the file named "test.c" from the current working directory. The object "test.c" is a *atomic* object; no derivation is needed to produce it. The command

```
test.c :run
```

requests the object produced by performing the run derivation on "test.c" (a *derived* object). Execution of this command would display the result of compiling and executing "test.c."

Given the starting and ending points of a derivation, Odin searches the derivation graph for a series of transformations (a path) which will produce the desired object. In the case of "test.c :run," Odin invokes the C compiler and the loader to create an executable object. That object is then executed and the output collected for display to the user. The process of compiling, loading, and executing "test.c" are performed invisibly. Intermediate objects, such as the object code, are derived automatically and cached so that they are available for use in filling future requests. The maximum amount of space to be used for storing derived objects may be specified by the user. When this limit is exceeded, derived objects are deleted using a least-recently-used strategy.

If intermediate objects are available, Odin will use them to perform the derivation in the "cheapest" way possible. If multiple paths from one object type to another exist, Odin will use a default path to resolve the ambiguity and avoid cycles.

When the user requests a derived object which is not currently in the database, Odin replaces macro names in a skeleton command script with the appropriate input and output host system files. The resulting command script is then invoked. This, in turn, invokes the tool that performs the appropriate transformation.

The user may also request that derivations be performed on compound objects. A compound object contains the names of the objects from which it is composed. When a derivation is requested, it is applied to these objects as a group, rather than

individually. If the contents of the compound object "myprog.ref" are

module1.c
module2.c
module3.c

the object

myprog.ref :exe

will contain the result of compiling and linking "module1.c," "module2.c," and "module3.c." This form of derivation corresponds to the view that an executable program is derived from the set of its individual components.

In addition to displaying objects, a user must also be able to modify objects. The modification of atomic objects from within Odin is performed through the use of a second type of command known as a *transfer* command. The transfer command is used to copy the contents of one object into another. Syntactically, the objects are separated by a right angled- bracket and the copy occurs in the direction implied by the arrow. For example,

test.c :fmt > good.fmt

places the result of formatting "test.c" into the file "good.fmt."

An extension of the transfer command makes it possible to interface to host system commands for use as "filters" or "editors." The command

test.c :fmt > :more

would display the result of running the system command "more" on the formatted source object derived from "test.c." ("more" is a Unix filter which displays a text file, one screenfull at a time, on a CRT.) The command

test.c > :emacs

would invoke the host system editor "emacs" on "test.c."

Odin also provides two forms of consistency maintenance among objects. The first form is similar to that of the Unix *make* utility [Feld79]. Objects in Odin are constrained to be consistent with respect to their creation/derivation times. If the object "test.exe" is derived from "test.c," in order for "test.exe" to be "consistent" with "test.c," the most recent derivation of "test.exe" must have occurred *after* the last modification of "test.c." When a request for "test.exe" is issued (e.g. "test.c :exe"), Odin consults its database. If "test.exe" already exists, its time of modification is compared with that of "test.c." If "test.c" has not been modified since "test.exe" was derived, no new derivation is necessary and none is performed. The existing object is simply returned from the database. If "test.exe" is out of date with respect to "test.c," a new derivation is performed.

The second form of consistency maintenance involves the use of "sentinels" [Clem84]. This is a more general mechanism that allows the specification of arbitrary semantic constraints. It is, however, weaker in that violations of these constraints are detected but not automatically corrected.

2.2. Extensions to Odin

To allow management of objects in the distributed system, some minor modifications and extensions to Odin were made. Modifications involved changes to the Odin code itself. These were kept to a minimum and were accomplished by adding new functions

in self-contained modules. The majority of Keystone's distributed capabilities were implemented using features already present in Odin. This was done by extending the derivation graph to provide types representing remote objects and adding new tools for manipulating these types. The following sections describe these modifications and extensions.

2.2.1. Modifications to Odin

Two new functions, *do_export* and *do_import,* were added to the Odin system. These functions are used to provide notification across the network when an object has been modified. This notification makes it possible for a node to notice when an object which it imports has changed and mark its local copy, as well as any objects derived from it, as out of date.

The do_export function is invoked when the user exits from Odin. If an exported object has been modified, do_export checks the export list to see who has received that object. Those nodes are then notified of the change. Currently, one node (the original owner) serves as the master node for an object. Remote nodes are notified of changes to an object only when they occur on the master node.

The do_import function is invoked when the user enters Odin. It checks a special, temporary file which contains the names of any imported objects which are out of date. Any such objects are then marked and the next access causes a new derivation (which necessarily involves an import).

Note that, since do_import is called only on entry to Odin and do_export is called only on exit, changes to global objects are not immediately visible to remote nodes. In principle, this could cause difficulty due to the use of out-of-date or inconsistent versions of objects. In practice, the granularity of notification which is required is not particularly fine and users seem to enter and exit Odin frequently enough that problems do not arise. If this proves to be a problem in other contexts, it can easily be addressed by having Odin call do_export and do_import more frequently.

2.2.2. Extensions to the Derivation Graph

The present version of Keystone relies heavily on several extensions to Odin's derivation graph. These extensions were made using capabilities already present in Odin. They were described using the specification language presented in Section 2.1.1.

The principal extension is the definition of a set of object types known as *remote_object_references.* There may be one remote-object reference corresponding to each atomic type in the derivation graph. Files of this type are denoted by adding the suffix ",x" to the file extension associated with the object's base type. For example,

```
test.c,x
```

denotes an object of type *remote_C_source_ref* and is a reference to a remote object of type *C_source.*

The name of the remote-object reference is intended to provide a "local" name through which the remote object may be accessed. For this reason the remote-object reference contains the global name by which the remote object is known over the network. The contents of a remote-object reference are thus a pointer to the actual file. For example, "test.c,x" might contain

sol:/toolpack/sutton/src/prog.c

where "sol" denotes a node and "/toolpack/sutton/src/prog.c" is the pathname of an object (file) on that node.

For purposes of importing and exporting objects it is unnecessary and probably undesirable to restrict global names to node/pathname combinations that indicate the actual location of an object. It is expected that objects will migrate and that they will exist in multiple copies. Under such conditions it would be difficult to maintain global names as actual network locations. In view of this, it seems most appropriate to treat global names as symbolic identifiers that bear no necessary relation to the locations of the objects they represent. In fact, the import/export mechanism does not depend on this correspondence. In the present implementation, however, the mechanism for notifying nodes of changes to imported objects does depend on the correspondence between the global name and actual location of an object. Consequently, for the current prototype, it is necessary to restrict global names as described above.

The derivation graph has also been extended to include derived types which may be produced from remote-object references. By convention, these are denoted by appending the prefix "imp_" to the usual type designation. Thus,

test.c,x :imp_run

denotes the imported object produced by compiling and running (on the remote machine) the file pointed to by "test.c,x."

Although intermediate objects, such as object code, are produced and cached on the remote machine, they are not exported. Only the requested object is exported. These objects are, however, available for local use on the remote machine or for subsequent exports without rederivation. Imports thus cause changes to the database which are not initiated by the local user. In this sense, the database may be said to be shared. Since Odin manages the objects in the database automatically, this is usually transparent to the user and is, at worst, only a minor inconvenience. In fact, there may be an advantage to the remote user since these objects are now cached and subsequent requests which involve them may be filled more quickly.

An "imp_c" derivation makes it possible to import the actual source object corresponding to "test.c,x." Once the source has been imported, derivations may be performed locally. For example,

test.c,x :imp_c :run

imports the source and compiles, loads, and executes it on the local machine. All intermediate objects are produced and cached locally.

Note that, due to the heterogeneous nature of the network, it is necessary to carefully specify certain objects in order to avoid nonsensical derivations. For example, the commands

test.c,x :imp_exe

and

test.c,x :imp_c :exe

will produce identical executable files if the local and remote machines are identical. However, in the first case, the compilation is performed on the remote node and the result is imported. In the second case, the source is imported and the compilation is

performed locally. If the two nodes have different instruction sets, the objects of type "exe" and "imp_exe" will be different.

2.2.3. Additional Tools

As described above, derivations may be performed on objects which are remote object references. In fact, importing an object is viewed as a derivation from a remote object reference. To perform these derivations, an *import tool* was added to Odin. The import tool receives (via shell variables) the name of the object from which the derivation is to be made, the derivation specification (e.g. ":imp_exe"), and the name of an object in the local database which will serve as the repository for the derivation. The import tool then invokes the importer process as described in Section 2.3.2.

2.3. Management of Imported and Exported Objects

As noted above, Odin provides management only for objects local to an individual workstation. Management of objects which are imported from or exported to other workstations is handled by a complementary pair of processes: the *importer* and the *exporter*. Both the importer and the exporter are long-lived processes which run in the background on each node.

The importer and exporter processes make use of three databases, a *translation table,* an *import list,* and an *export list,* to store the information required to manage objects in the distributed system. The following sections describe these databases and the functions of the importer and exporter processes. Figure 3 illustrates the flow of control among the databases, the importer and exporter processes, and Odin.

2.3.1. The Databases

The *translation table* contains pairs of corresponding local and global object-names. Given either a local or global name for an object, the translation table can be used to obtain the other. Local names may be absolute or relative (to the current working directory) pathnames for files on the local node; they are known only on the node to which they apply. Global names are identifiers which are shared among all nodes in the network that may have access to the corresponding object.

Local names may denote objects that are references to remote objects (see discussion of remote-object references, in Section 2.2.2). It is not necessary for the user to explicitly create these remote-object references. They are created automatically when the local/global name pair for such objects is entered in the translation table. Since the remote-object reference should contain the global name of the object it is simple and appropriate to automate the creation of these references in this way.

The *import list* specifies those objects which may be obtained from other nodes. In particular, an entry in the import list contains the global name of an object, the name of a node that may export that object, and the locations of the Odin directories on both the local and remote nodes. Since a given object may be obtainable from more than one node, it may appear in the import list more than once (although each time with a distinct node name). The global object names and node names are used by the importer process when attempting to import an object. The locations of the Odin directories are used to set up files that are used by the do_import and do_export functions (Section 2.2.1).

The *export list* specifies those objects which have been provided to other nodes. Each entry in the export list contains the global name of an object and the name of a node to which it has been exported. Since an object may be exported to multiple nodes, it may appear several times in the export list. Entries are added to the export list by the exporter process whenever an object is exported.

An additional database will be needed to specify access control for objects in the distributed environment. The levels of control are intended to be more than simple read/write access. For example, the user of a given node should be able to specify that a particular object is exportable only if the current load average on that node is below a certain limit [Clem85]. This database is planned, but has not yet been implemented.

Each Keystone process also maintains two other lists of information to which it has exclusive access. These include the *node list* and *connection list*. The node list contains information describing each node in the network, including the conventional "port numbers" used to establish connections with the Keystone and Odin processes on those nodes. This information is the same for all processes since the network is relatively fixed, and it could be kept in a shared database. However, the node list is a small structure, and once it is initialized it is only read, so for efficiency each process keeps its own copy in memory. The connection list contains information and buffers used by a process in managing its connections to other processes. These connections are unique to the process and may change with every transaction.

2.3.2. The Importer

The *importer* process manages the importing of objects from remote nodes. The importer may be invoked in response to a request for an "import" derivation. If the request is for an object which has been previously imported, that object will have been cached locally. In that case, the local version will be returned unless it is out of date and the importer will not be invoked. The importer is actually invoked whenever the object of the import is not available locally or the local copy of the object is known to be out of date.

When an import derivation is performed, information necessary to perform the derivation is passed to an import command script. This information consists of the name of the local atomic object from which the derivation is to be made (a remote-object reference), the derivation specification, and the name of the object in the local database into which the derivation is to be deposited. The import command script extracts the global name from the remote-object reference and passes the global name, together with the derivation specification and local destination, to the import tool. The import tool packages this information into the Keystone message format, establishes a connection with the importer, and sends it the import-request message.

When the importer receives a request, it uses the global name to determine, from the local import list, which nodes export the object. The importer then sends each potential exporter a request for information on the cost of obtaining the object (see the discussion of costs, Section 2.3.3). These requests contain the global name of the object and the derivation specification. When the costs have been received from all of the exporters (or when a time-out period has expired), the importer examines the costs returned and requests the object from the least expensive source (if any of the nodes can in fact supply the object). Object requests also include the global name of the object and the derivation specification.

The importer may request an object from a remote exporter only to find that the exporter cannot supply the object. This might occur if, for example, the node went down between sending the cost information and actually supplying the object itself. In that case the importer tries to obtain the object from the least costly of the remaining sources, if any.

The importer may be unable to identify a node that can supply the requested object, either because the export list includes no exporters for the object or none of the listed exporters is currently able to provide it. Currently, the request will fail under these circumstances. Future versions of Keystone will include provisions for polling other nodes to find other suppliers.

If the importer successfully obtains the object, it is deposited in the local Odin database. In any case the importer returns a message to the import tool that indicates the success of the derivation. The import tool then exits with the appropriate status, which is captured by the import command, which, in turn, provides Odin with error or warning messages as needed.

2.3.3. The Exporter

The *exporter* process manages the exporting of local objects to remote nodes. The exporter responds to requests from remote importers; it does not communicate, even indirectly, with the user's local copy of Odin. To fill remote requests, the exporter does, however, create its own (temporary) Odin processes.

The exporter receives two principal types of request from importers: requests for cost information and requests for objects (see below). With both types of request the exporter receives the global name of the object and a derivation specification. It then translates the global name into a local name using the translation table. It also converts the import derivation to the corresponding local derivation which, by convention, can be determined by deleting the "imp_" prefix from the import derivation. If the global name cannot be found in the translation table then the requesting importer is sent an INFO_REQUEST_ERROR message and the request is not otherwise answered.

If the request is for cost information the exporter carries out the following series of actions. First, it invokes an Odin process to check whether the object exists (or can be derived) locally. If the object is a derived object that is not already in the local Odin database (or is in the database but is out of date) then the check for existence forces a new derivation of the object. If the object does then exist, the exporter returns a cost of AVAILABLE. If not, the exporter searches its own import list for another node from which it may be able to import the object. If it finds such a node it returns a cost of IMPORTABLE. Otherwise a cost of NOT_AVAILABLE is returned.

This cost is actually an index of the "effort" required on the part of the exporter to export the object. It is not a measure of something to be paid by the importer, except that the importer may have to wait longer to obtain the object from a more costly source than from a less costly source. At present only the cost values described above are used in Keystone. A cost of NOT_AVAILABLE indicates that the remote node does not have the requested object and does not know of any node from which it could import the object. A cost of IMPORTABLE indicates that the remote node does not have the requested object but does know of one or more nodes from which it may be

able to import it before in turn exporting it to the requester. A cost of AVAILABLE indicates that the requested object is directly available on the remote node. The importer does not request an object from any remote node that returns a cost of NOT_AVAILABLE. The importer may request an object from a node that returns a cost of either AVAILABLE or IMPORTABLE, although AVAILABLE is considered to be a lower cost than IMPORTABLE and is chosen preferentially.

The exporter must perform a similar series of actions in response to requests for objects since there is no guarantee that the information previously supplied to the requester is still current (or even that the requester has previously asked for information, although checks for this could easily be implemented). Checks are again made to determine whether the object exists locally or is importable. If the object does not exist locally and is not importable, then an OBJECT_REQUEST_ERROR message is returned. If the object exists then the exporter returns it directly. Otherwise, if the object is importable, then the exporter invokes a temporary Odin to obtain the object from yet another node, and then returns it to the original importer.

The export mechanism described here allows for transitive requests which may span several nodes. The transitive nature of a request is transparent to the requester except for the additional time required to establish multiple node-to-node connections and move the object. It is possible that a transitive request could generate a cycle (i.e. A requests an object from B, B requests the object from C, and C requests it from A). To avoid this, each requesting node in the sequence adds its own name to a *request list* that is included with the request. No node that has already been asked for the object as a part of the current derivation will be asked for it a second time.

Whenever an object is exported, the exporter adds an entry to the export list. These entries specify the object that was exported and the node to which it was exported.

3. Experience with Keystone

As mentioned, two successive prototype versions of Keystone have now been implemented. Both versions run on a network consisting of Sun workstations, a Vax, and a Pyramid. Both were implemented on top of Berkeley Unix, version 4.2, and both make use of the interprocess communication mechanisms provided by this operating system. Neither version required any changes to the standard Unix operating system. The first prototype was implemented using a combination of C language routines to manage the import and export lists and shell scripts to perform importer and exporter functions. For the second prototype, the importer and exporter were implemented directly in C to improve the speed of network requests.

One of the principal goals of the Keystone project is to study cooperation and sharing of information in a distributed software environment. Accordingly, we are particularly interested in coordination tools, tools which facilitate cooperation among individuals working on a software project.

Typical coordination tools include tools for the automated construction of programs from their component parts (configuration management) and tools for tracking and control of modifications (version control or history management). While some would lump both sets of tools together under the general category of "configuration management" (see e.g. [Baze85]), it is useful to treat them separately here. Configuration management and version control were chosen as research vehicles for the current version of

Keystone because they offer the opportunity to explore the extension of both basic environment capabilities and user-defined tools to the distributed context. Configuration management in Keystone involves distribution of Odin's capabilities for performing derivations on compound objects. Version control involves the addition of external tools to the environment using the framework established by remote-object references and the import tool.

In this section we describe our experience in extending these tools to a distributed environment using the object management paradigm embodied in Odin/Keystone.

3.1. Configuration Management

For the purposes of this discussion, we will consider configuration management to be the automated generation of a software system from its component parts. The essential features of configuration management are the ability to specify the components of a system and the ability to automatically generate the system from that specification.

3.1.1. Implementation

Several different versions of configuration managers have been implemented, most of them traceable to the Unix *make* utility [Feld79]. Make enables the user to specify the configuration of a software system in terms of the binary files from which it is composed. This specification is described in a *makefile* which also lists the various components upon which each binary file depends. Once a system has been built using *make,* it is rebuilt only if one of the components has changed. This is accomplished by comparing the date and time of modification for each component specified in the *makefile.* If no components have changed, the system is not rebuilt. If some components have changed, only those which have been modified are recompiled. This approach insures that the system is up to date and saves processing time by only recompiling those components which are no longer current. Because *make* always uses the most recent version of a component, if multiple versions of a system are desired, each must be specified in a separate *makefile.*

Keystone includes comparable capabilities for configuration management. These capabilities arise naturally from Odin's ability to specify and perform derivations on compound objects (objects composed of multiple simple files). Configuration management is accomplished by using "reference" files to specify the names of component source files that make up the compound source object for a system as described in Section 2.1. As with *make,* subsequent requests for an executable object will result in a new derivation only if one of the components has changed since the last derivation. If a new derivation is required, it will involve only those components which have changed.

While the use of reference files provides a capability comparable to that of make, several differences are readily apparent. The most obvious difference is that reference files list the source objects which make up the system rather than binaries as in a makefile. Another important difference pertains to the amount of information which must be used to describe a system. With make, it is necessary to explicitly describe all dependencies for a given object. If other files are to be included at compile time (e.g. "#include"s in C) these must be specified as part of the dependency information. In Odin, these dependencies are computed (when necessary) at run-time using tools specified in the derivation graph. Thus, the user need only specify the simple objects which make up a system.

The example presented in Section 2.1 assumes that all of the components of "myprog" are available locally. If one or more of the objects must be imported, this will be reflected by the presence of remote-object references in the .ref file. For example, if both module1.c and module3.c are imported from other nodes, "myprog.ref" will contain

```
module1.c,x :imp_c
module2.c
module3.c,x :imp_c
```

By specifying the ":imp_c" derivations of "module1.c,x" and "module3.c,x" we insure that the source object will be imported and compiled locally, avoiding potential problems from importing code compiled on a foreign (and possibly different) machine. Since each node which imports an object is notified in the event of changes, the derived object will be consistent with its components.

3.1.2. Comparison with Other Environments

Keystone's configuration management facilities are implemented using Odin's inherent capacity to perform derivations on compound objects. Management of objects in the distributed system is accomplished using standard import derivations. Our experience indicates that this mechanism provides a straightforward and logical extension of Odin's capabilities to the distributed environment. Configuration management in Keystone may be compared with that in two other environments which support software development in a distributed context: Cedar [Teit85] and the Domain Software Engineering Environment (DSEE) [Lebl84].

Configuration management in Cedar is accomplished using the *System Modeller* [Schm82], [Lamp83a]. The Cedar programmer describes a piece of software by writing a *system model* in *SML,* a module interconnection and system description language [Lamp83b]. The system model specifies: the versions of the modules which make up the system together with some information about their locations in the distributed file system; the interconnections among the modules; and additional information needed for compiling and loading the system. The System Modeller then automatically builds an executable form of the software by compiling and loading the appropriate modules. The Cedar editor notifies the System Modeller when a new version of a source module is created and this change is automatically incorporated into the system model. As in make and Odin, only those modules which have changed are recompiled.

Configuration management in DSEE is performed by the *Configuration Manager.* As in Cedar, the configuration for a system is specified in a *system model* using a *system model language.* In DSEE, however, the description of which components (modules) comprise the system is distinguished from the *configuration thread,* a description of which version of each component is to be used in actually assembling the system [Lebl85]. Prior to performing a build, the system model and configuration thread are used to generate a *bound configuration thread* based on rules specified by the programmer. The bound configuration thread contains the specific versions which are to be used in assembling the system. It is thus possible to construct several versions of a system from a single DSEE system model.

Our intent in studying configuration management in Keystone was not to develop a highly sophisticated configuration management tool. The purpose of this work was to determine the feasibility of distributing Odin's basic capabilities by using extensions to

the derivation graph in conjunction with the import tool. The configuration management facilities provided by Keystone are, however, comparable to those found in Cedar although less extensive and less flexible than those provided by DSEE. In addition, the Keystone approach supports the existence of multiple copies of a module. A system may be constructed from any of the available copies. This feature has no counterpart in either Cedar or DSEE. Finally, other approaches to configuration management are possible within the Odin/Keystone framework. Tools which meet special needs may developed and added to the environment by specifying new entries in the derivation graph. In particular, capabilities which are equivalent to those of DSEE could be provided in this fashion.

3.2. Version Control

In a software environment, it is necessary to manage revisions to modules or groups of modules. This is especially true in cases where more than one individual may be involved in the development of those modules. Access to objects for update purposes must be controlled and multiple, simultaneous updates must be prevented. Recording the nature of changes as well as their rationale is also an important aspect of version control.

3.2.1. Implementation

Again, several different change or revision management systems have been implemented. Most of these are variations on the theme established by SCCS [Roch75] and RCS [Tich82]. These utilities represent all versions of an object in a special file. While SCCS and RCS differ in implementation details, both store revisions as incremental changes or "deltas," rather than storing each version in its entirety, to save space. When a particular version is requested, it is generated by applying the appropriate deltas to the base object.

Both SCCS and RCS use a reserve/replace discipline in which a user wishing to modify a file obtains a separate copy of that file for editing. Once a file has been checked out for modification, additional writeable copies may not be obtained. This prevents simultaneous updates to a file. When the changes are complete, the user places the back in the management system, thus creating a new version. When the file is replaced, log messages may be included to describe the changes which were made and the reasons for making them.

Under the Odin/Keystone approach, version control is viewed as a capability which may be added via tools provided by the user. To manage changes to objects in the distributed system, Keystone uses RCS as an external tool. The decision to use RCS was made because RCS is readily available and familiar. It also provides an array of capabilities which make it possible to explore various aspects of object management in Keystone. The use of RCS made it unnecessary to construct a special version control tool for investigation in the distributed environment. However, other tools, such as SCCS, could also have been used or new tools with different capabilities could have been built.

The basic commands available in RCS are "checkin" (ci), "checkout" (co), and "get log info" (rlog). The checkin command is used to place a file under RCS control and to deposit modifications back into the system. When a file is checked in, the user is prompted for a log message. The checkout command is executed to obtain a copy of the current (or any other) version of the file. The copy may be read only or, if

modifications are to be made, a writeable copy may be obtained. If the copy is writeable, the RCS file is locked, preventing other users from also checking out writeable copies. Additional read-only copies may still be obtained using checkout, however. The rlog command displays log entries as well as other information about the revisions to a file.

As described in Section 2.1, there are two kinds of tools that can be used from Odin/Keystone: "derivation tools" and "editor tools." A derivation tool is one that reads a set of input files and produces a set of output files. It cannot modify its input files or any global information. Derivation tools cannot read any global information and they cannot be interactive. All other tools are editor tools. The use of derivation tools is specified in the derivation graph while editor tools are invoked directly by the user via the "> :" operator.

The RCS tools "co" and "rlog" are both derivation tools and are thus specified in the derivation graph. Since "ci" produces a change in and atomic object (the RCS file), it is an editor tool and is therefore invoked using the "> :" operator.

Implementation of distributed versions of co and rlog in Odin/Keystone was straightforward. Two derivations from RCS files were already available in Odin: ":c," which performs a checkout, and ":log," which performs an rlog. These commands simply invoke the appropriate RCS function, depositing the result in the corresponding object in the Odin database. Extension of these commands to the distributed system required only extension of the derivation graph and addition of appropriate command scripts as was done for other import derivations.

Implementation of the distributed checkin command is substantially more difficult. In order to accomplish this, it would be necessary to write an "editor" script to perform a distributed checkin. This script might be named "dci" and would be invoked with the request

object > :dci

The dci script needs to perform the following four functions:

(1) Look up the global name of the RCS file from which "object" was derived.

(2) Determine the new version that is to be stored in "object."

(3) Transfer "object" to the remote node.

(4) Perform a checkin on the remote node.

The additional difficulty in implementing a distributed checkin is due to the difference in the ways in which Odin treats changes to derived and atomic objects. The inclusion of derivation tools such as co and rlog is supported quite naturally within this framework. The use of editor tools for operations like checkin is logically consistent with Odin's approach to changing atomic objects. But, development of appropriate command scripts can present user interface difficulties as, for example, in specifying options on the dci command or providing log messages. The design of a command script with an appropriate user interface is still under study.

Change management and configuration control can be integrated quite easily in Odin/Keystone. In specifying the compound object corresponding to a system the user simply names the RCS file (denoted by the extension ",v") and specifies the appropriate derivation. Such an object might have the following contents

```
module1.c,v :c
module2.c,x,v :c
module3.c,v :c
```

Here, module1 and module3 are local RCS files module2 is a remote object which is also under RCS control. Although it is not shown in this example, the checkout derivation is also capable of accepting parameters which specify the particular version of the source module which is to be produced.

3.2.2. Comparison with Other Environments

It is also useful to compare change management capabilities in Keystone with those in Cedar and DSEE.

Cedar does not provide special facilities for source code control. In fact, Cedar is based upon a model of software development in which only one individual has responsibility for developing and testing a given module. Thus, coordination tools of this sort are not needed.

Cedar does, however, allow several versions of a module to exist at any time. This is accomplished by considering each file in the Cedar file system to be a unique, immutable object characterized by a unique identifier (UID). Thus, when a file is modified to create a new version, a new object, with its own UID, is created. In Cedar, each version is stored in its entirety, in contrast to the use of incremental changes (deltas) as in SCCS and RCS. As noted above, the Cedar editor notifies the System Modeller when a new version of a source module is created and this change can be incorporated into the system model.

Source code control in DSEE is performed by the *history manager*. The history manager is similar to SCCS and RCS in that the user reserves a file for editing and a new version is created by replacing the file when the modification is complete. When the module is replaced, additional information, such as the time of modification, the reason for the change, and the name of the person making the change are also saved. When a new version of an object is created, the local history manager notifies managers on other nodes.

Keystone, then, provides version control support comparable to that of DSEE. Again, however, our intent was not to implement a specialized version control system but rather to investigate the issues associated with adding tools which require the distributed use of editor-type operations. While there are clearly some implementation-level issues to be resolved, we expect that it will be possible to add tools which provide appropriate support.

4. Discussion

Development and use of the Keystone prototypes has provided valuable insight into the utility of the basic approach used for this research. This work has also helped to define and clarify a number of issues concerning object management in a distributed software environment. This section discusses several observations based on our experience with Keystone.

4.1. Object Management

Experience with Keystone has demonstrated that an object management system based on local object managers augmented with capabilities for importing and exporting objects is indeed a viable approach for distributed software environments. Object management in such a distributed environment will, however, depend strongly on the capabilities of the local environment. This is especially true with respect to the ease with which basic operations may be distributed and control over persistent objects. Our experience with these aspects of the Odin/Keystone environment is discussed below.

4.1.1. Distribution of Operations

Distribution of the derivation capabilities of the Odin environment has been straightforward. Implementation has required only the addition of *importer* and *exporter* processes, and an *import tool,* together with data management facilities to keep track of objects in the distributed system. Implementation of distributed derivations required only minimal changes to the local object manager.

Distribution of "editor" functions is substantially more difficult (see e.g. Section 3.2 for a discussion of the difficulties encountered in implementing a distributed RCS "checkin" command). Distributed editor operations require more complicated mechanisms for the exchange of objects and information between nodes than do distributed derivations.

One approach to solving these problems is the addition of new, user-level tools to perform distributed editor functions. The existence of "local" and "distributed" versions of tools is undesirable, however, because it forces the user into two different modes of interaction. The need to develop distributed counterparts of common tools also reduces the extensibility of the environment. An alternative would be to extend the basic capabilities of the environment to allow better integration of editor tools.

4.1.2. Control Over Objects

Object in Odin/Keystone are standard files in the host file system. Thus, they generally outlive any invocation of the tools that create and use them, or even of the Odin environment itself. Odin provides varying degrees of control over these objects in the local environment.

Control over atomic objects is loose. Atomic objects are created by the user, typically using "editor" tools, and these objects exist in the user's directory, rather than in the Odin database. Atomic objects are therefore easily accessible both from within Odin and from user operations performed outside of Odin. Consequently, it is possible for the user to inadvertently modify an atomic object, making it inconsistent with its original specification. The advantage of this loose control is that a user can experiment with a variety of mechanisms for creating and modifying atomic objects. Whether this flexibility can be preserved while exerting tighter control over atomic objects is an open question.

Control over derived objects is stronger. Derived objects are created systematically within the Odin system and are stored automatically in the Odin database. While derived objects are still host system files, they are stored in special directories under names known only internally to Odin. Thus, it is difficult for a user to alter or otherwise use derived objects outside of Odin.

4.2. Extensibility

One of the principal goals of the Keystone project is to construct an extensible distributed software environment. Extensibility goes well beyond the ability to configure an environment to accommodate individual preferences (e.g. aliasing). Extensibility is the "ability to modify an environment in response to changes in the ways that the environment will be used" [Ridd85].

The Odin system provides a general mechanism by which users may incorporate arbitrary tools into an environment. Odin provides the ability to structure relations among tools and objects; otherwise its capabilities derive principally from the tools included by users. Tools to meet specific needs can be developed independently and may be added without restructuring the environment. The object-oriented command language provides a straightforward means of external (user-level) integration of these tools. Internal integration is achieved by allowing tools and tool fragments to communicate via host system files. This communication is controlled by the specifications in the derivation graph.

The Odin approach provides a high degree of extensibility for a centralized environment. One of the principal questions to be answered by the Keystone project is whether the Odin approach to extensibility can be straightforwardly adapted to a distributed environment. Consequently, in the initial prototypes, our goal was to modify Odin as little as possible and implement most of the capabilities for managing distributed objects via the derivation graph.

As described above, processes for importing and exporting objects are provided. An *import tool* which invokes these processes in response to a request for a remote object is also provided. This tool is called by user command scripts to accomplish the actual importing of an object. Given the import tool as part of the Keystone system, the user can add an import derivation in a manner exactly analogous to the addition of a new derivation in a single-node Odin system. The derivation graph is simply extended to include the new object type and the corresponding command script (which includes the import tool) is created. Once this has been done, there is no need to use special commands to accomplish an import derivation. This has the advantage that the tool used for the single-node derivation may be used for the distributed derivation without modification. Thus, the implementation of distributed derivations has been straightforward and has not required new user-level tools.

As noted above, the distribution of editor tools is more difficult and presently requires the development of new user-level tools. The need to develop new, specialized tools means that these capabilities are not easily extensible by users. Mechanisms that might support this extensibility are still under investigation.

4.3. Object Typing

Work on Keystone has highlighted two central issues associated with object typing: persistent objects and the influence of heterogeneous resources. Strength of typing is an issue in all software environments; approaches to this problem will, however, be affected by the distributed nature of an environment, particularly if multiple object managers are involved. Problems due to differences in resources from machine to machine in a network, such as the existence of several different CPUs, can be avoided by simply requiring the system to be homogeneous. However, since one of the principal

advantages of a distributed environment based on individual workstations is the ability to tailor the workstation to the task which it will perform, such a requirement is overly restrictive.

4.3.1. Persistent Objects

Atomic objects on Odin are strongly typed. However, since atomic objects are created by the user, the user must insure that the type of the object is properly indicated by the extension to the file name which is recorded in the derivation graph. Since Odin objects are persistent (i.e. they have lifetimes which are significantly longer than a single invocation of Odin), they are not always under Odin's control. Thus, even if an atomic object is created and identified properly, it may still be edited in such a way as to become inconsistent with its type indication. There is no provision for the checking the types of atomic objects. If a derivation is applied to an incorrectly-typed atomic object then a runtime error will result.

Derived objects are more strongly controlled. As noted in Section 2.1, the user is responsible for insuring that the operations defined in the derivation graph are appropriate for a given type of object. If a derivation defined in the derivation graph is not appropriate for a given type of object then Odin will still attempt the derivation, again producing a runtime error. Odin will, however, prevent any derivation that is not defined in its derivation graph. If the derivation graph has been correctly specified then type conventions will be enforced for any derivation performed from within Odin. Of course, since Odin objects are files in the host system, it is possible to (inadvertently or intentionally) circumvent these restrictions by performing some operations outside of Odin. As discussed above, however, it is difficult to access derived objects from outside Odin.

Keystone extends the Odin approach to typing to a distributed context. As in Odin, there is no control over the typing of atomic objects. This creates additional problems in Keystone because remote-object references are atomic objects, and there is no guarantee that a ",x" object in fact contains a remote-object reference. In addition, the type of an imported atomic object is not constrained to correspond to the type of its remote-object reference (a type mismatch could occur, for example, if the remote object were incorrectly typed).

As in Odin, the type of derived objects is more strongly controlled. Successfully imported derived objects are guaranteed to be of the type indicated by the derivation and may be used accordingly in future derivations. Furthermore, since imported objects are stored in the Odin database, imported atomic objects are maintained like derived objects. Thus they are relatively protected from capricious changes on the importing node.

The distribution of an environment poses some problems for the typing of objects that do not arise in a single-machine environment. A major concern is consistent type definitions for objects that may be shared between nodes; this problem is compounded by the existence of multiple object managers. Another concern is differences in resources between machines that may affect the equivalence of objects that are ostensibly of the same type. This problem is discussed in more detail in the next section.

4.3.2. Heterogeneous Resources

The existence of heterogeneous resources across the network creates additional problems in specifying object types and manipulating their instances. CPUs are perhaps the most important resource that may vary from machine to machine. Differences in CPU types will strongly affect the meaning of "type" for certain objects. Another example is the versions of software tools, such as compilers or debuggers, that are used to derive or access objects. The potential problems introduced by differing types of CPUs are discussed below.

While some derivations (e.g. formatting source code) are machine independent, others (e.g. compilation) are not. Thus, an object of a given "type" which is derived on one machine may not have the same attributes as an object of the same "type" derived on a machine with a different CPU. The approach used in Keystone is to have the user specify whether the derivation is to be performed locally or remotely. Other, similar approaches, such as a default derivation site which may be overridden by the user, may also be employed. It may also be desirable to have a more general mechanism which allows the user to specify the machine (or CPU type) on which a given derivation is to be performed. This approach can be extended to other types of resources, as well.

4.4. Logical View of the Environment

A physically distributed environment may present a user view which is logically centralized or logically distributed. The notion of a logically central repository for project information [Oste81] has become an established principle for environment design. Recently, however, this concept has been re-examined [Ridd86]. Many researchers now feel that it may, in fact, be desirable to have several logically distinct information repositories.

The current implementation of Keystone presents a logically distributed view to the user. The external nature of an object is apparent because it must be accesses via a remote-object reference whose name includes the ",x" suffix. To obtain access to a remote object, the user need only know its global name; there is no need to keep track of the physical location of the object. From the user view, then, Keystone partitions the information repository into two logical classes: local objects and remote objects.

The question of whether to present a logically centralized or distributed view to the user is still an open one. The utility of a centralized view is well established. It may, however, be desirable to provide special subnetworks for different functions, such as maintenance, and provide several logically separate databases which correspond to these functions. If the host network is to be heterogeneous, it will be difficult (and probably undesirable) to hide the distributed nature of the environment from the user. The answers to these questions are also likely to be context dependent. It may also be desirable to support different views for different purposes, in different organizational settings, or at different points in the software process. In these cases, a flexible approach will be necessary. The resolution of these questions will require additional experience with both centralized and distributed views in a variety of settings.

5. Summary

This report has described object management in the Keystone distributed software environment. Keystone was developed to explore issues which arise when the various

functions of software development are distributed over a (possibly heterogeneous) network of workstations.

The research reported here has investigated the feasibility of managing objects in a distributed software environment by augmenting an existing, single-machine object manager (Odin) with facilities for exporting/importing objects to/from remote nodes. This approach corresponds to creating a "federation" of local object managers in which responsibility for local objects remains local and users are able to specify the objects they are willing to share as well as the circumstances under which access to those objects may be obtained. An object manager based on such an architecture offers several potential advantages, chief among them the ability to use existing tools rather than develop new tools for the distributed environment and flexibility both in configuring the environment and in establishing cooperation among users.

Keystone provides distributed versions of Odin derivations. Users may import both atomic and derived objects from remote nodes using the standard Odin command syntax. Once imported, these objects become part of the local information repository and may be used in the same manner as locally derived objects. Our experience indicates that this capability is a natural extension of derivations to the distributed context and that the present architecture supports it well. By providing a simple import tool, it has been possible to allow users to easily add their own import derivations, thus preserving the extensibility provided by the Odin environment.

Tools which alter remote atomic objects ("editor" tools) have proven more difficult to distribute. These tools require more complex exchanges of information between nodes and, in some cases (e.g. RCS) require interaction with the user. Suitable approaches for distributing these types of functions are still being explored.

Our experience with Keystone indicates that the architecture described here represents a viable approach to distributed object management for distributed software environments. Future work on the Keystone project will make use of the extensibility and flexibility provided by this architecture to explore other issues in distributed software environments. This work will have two principal thrusts. The first will be continued enhancement of the capabilities of the system itself. The second will be evaluation of the system through use on actual software development projects to determine both the utility of the Keystone architecture and the impact of the distributed environment on the software process.

6. Acknowledgements

Lee Osterweil has provided much of the motivation and support for the Keystone project. The initial design for Keystone evolved through discussions with Dennis Heimbigner and Lee Osterweil.

7. References

[Baze85] R. Bazelmans, "Evolution of Configuration Management," *Software Engineering Notes,* vol. 10, no. 5, pp. 37-46, 1985.

[Clem85] G. M. Clemm, D. M. Heimbigner, L. J. Osterweil, and L. G. Williams, "KEYSTONE: A Federated Software Environment," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large,* Harwichport, Massachusetts, June 1985, pp. 80-88.

[Clem84] G. M. Clemm, "Odin: An Extensible Software Environment," Technical Report CU-CS-262-84, University of Colorado, Department of Computer Science, 1984.

[Feld79] S. I. Feldman "Make - A Program for Maintaining Computer Programs," *Software Practice and Experience* vol. 9, pp. 255-265, 1979.

[Heim85] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Mangement," *ACM Transactions on Office Information Systems,* vol. 3, no. 3, pp. 253-278, 1985.

[Lamp83a] B. W. Lampson and E. E. Schmidt, "Organizing Software in a Distributed Environment," *Proceedings of the SIGPLAN 83 Symposium on Programming Language Issues in Software Systems,* published as *SIGPLAN Notices,* vol. 18, no. 6, pp. 1-13, 1983.

[Lamp83b] B. W. Lampson and E. E. Schmidt, "Practical Use of a Polymorphic Applicative Language," *Proceedings of the 10th Symposium on Principles of Programming Languages,* pp. 237-255, 1983.

[Lebl85] D. B. Leblang and G. D. McLean, Jr., "Configuration Management for Large-Scale Software Development Efforts," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large,* Harwichport, Massachusetts, June 1985, pp. 122-127.

[Lebl84] D. B. Leblang and R. P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* published as *Software Engineering Notes* vol. 9, no. 3, pp. 104-112, 1984.

[Oste81] L. J. Osterweil, "Software Environment Research Directions for the Next Five Years," *Computer,* vol. 14, no. 4, pp. 35-43, 1981.

[Ridd86] W. E. Riddle and L. G. Williams, "Software Environments Workshop Report," *Software Engineering Notes,* vol. 11, no. 1, pp. 73-102, 1986.

[Ridd85] W. E. Riddle and J. C. Wileden, "Environment Extensibility," Technical Report SDAM/19, software design and analysis, inc., Boulder, Colorado, 1985.

[Roch75] M. J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering,* vol. SE-1, no. 4, pp. 364-370, 1975.

[Schm82] E. E. Schmidt, *Controlling Large Software Development in a Distributed Environment,* Ph.D. Thesis, EECS Department, University of California, Berkeley, 1982 and Technical Report CSL-82-7, Xerox PARC, 1982.

[Teit85] W. Teitleman, "A Tour Through Cedar," *IEEE Transactions on Software Engineering* vol. SE-11, no. 3, pp. 285-302, 1985.

[Tich82] W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings of the Sixth International Conference on Software Engineering,* pp. 58-67, 1982.

[Zdon85] S. B. Zdonik and P. Wegner, "A Database Approach to Languages, Libraries and Environments," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large,* Harwichport, Massachusetts, June 1985, pp. 89-112.

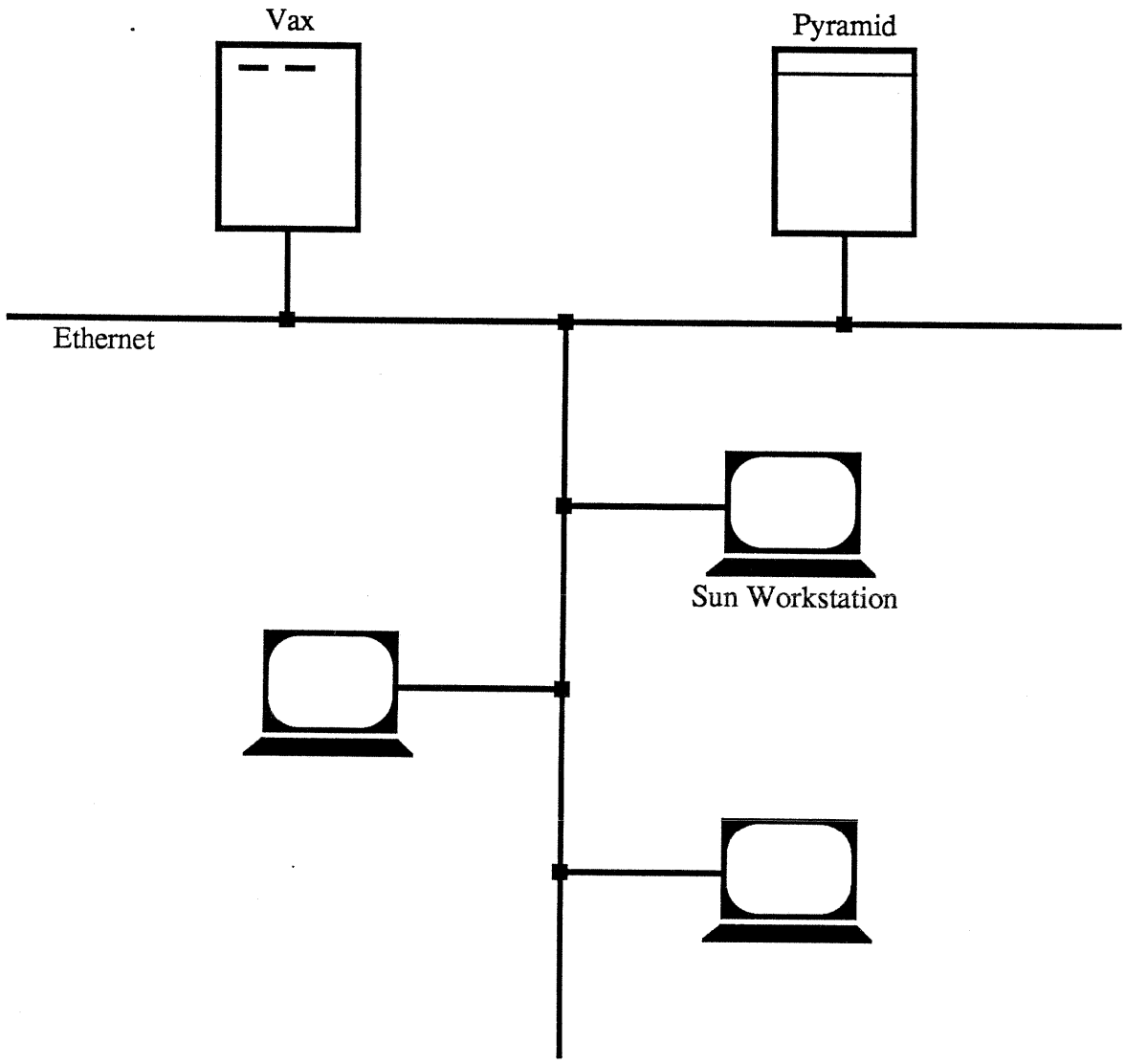


Figure 1: The Keystone Network.

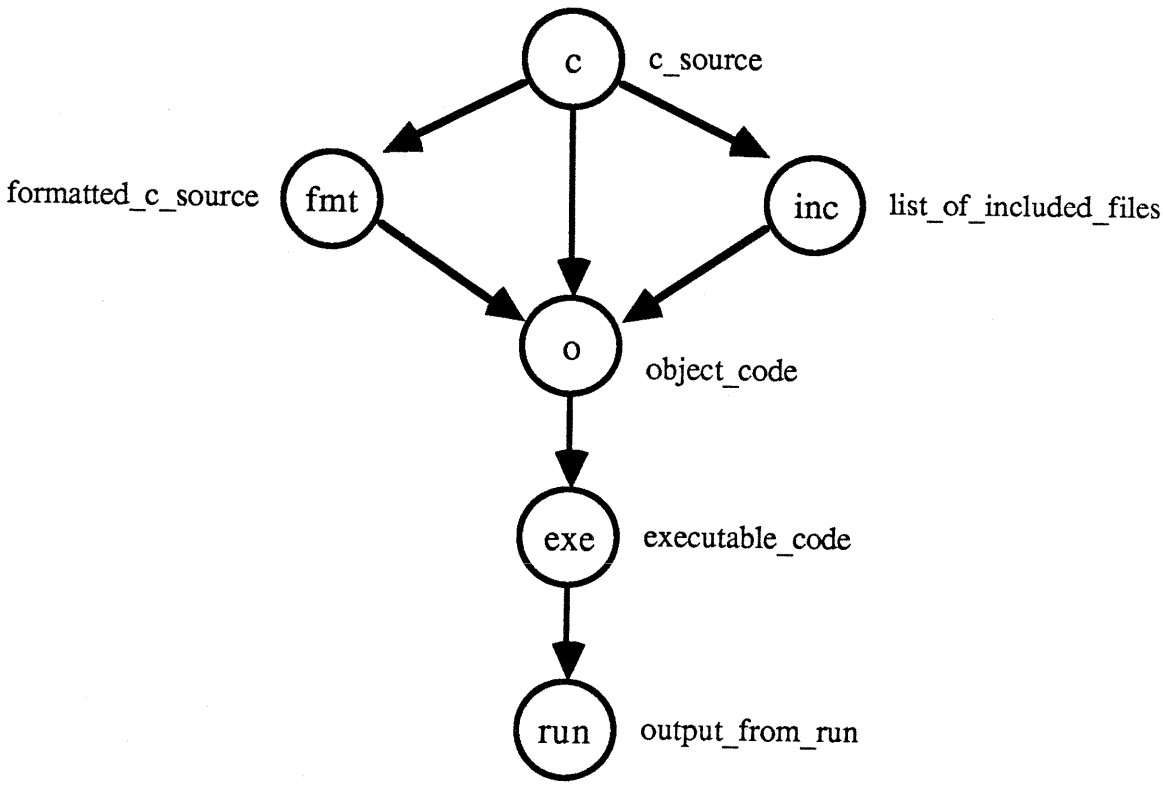


Figure 2: Example Odin Derivation Graph. Each node represents an object type and each edge represents a transformation which derives the object at its head from the object at its tail.

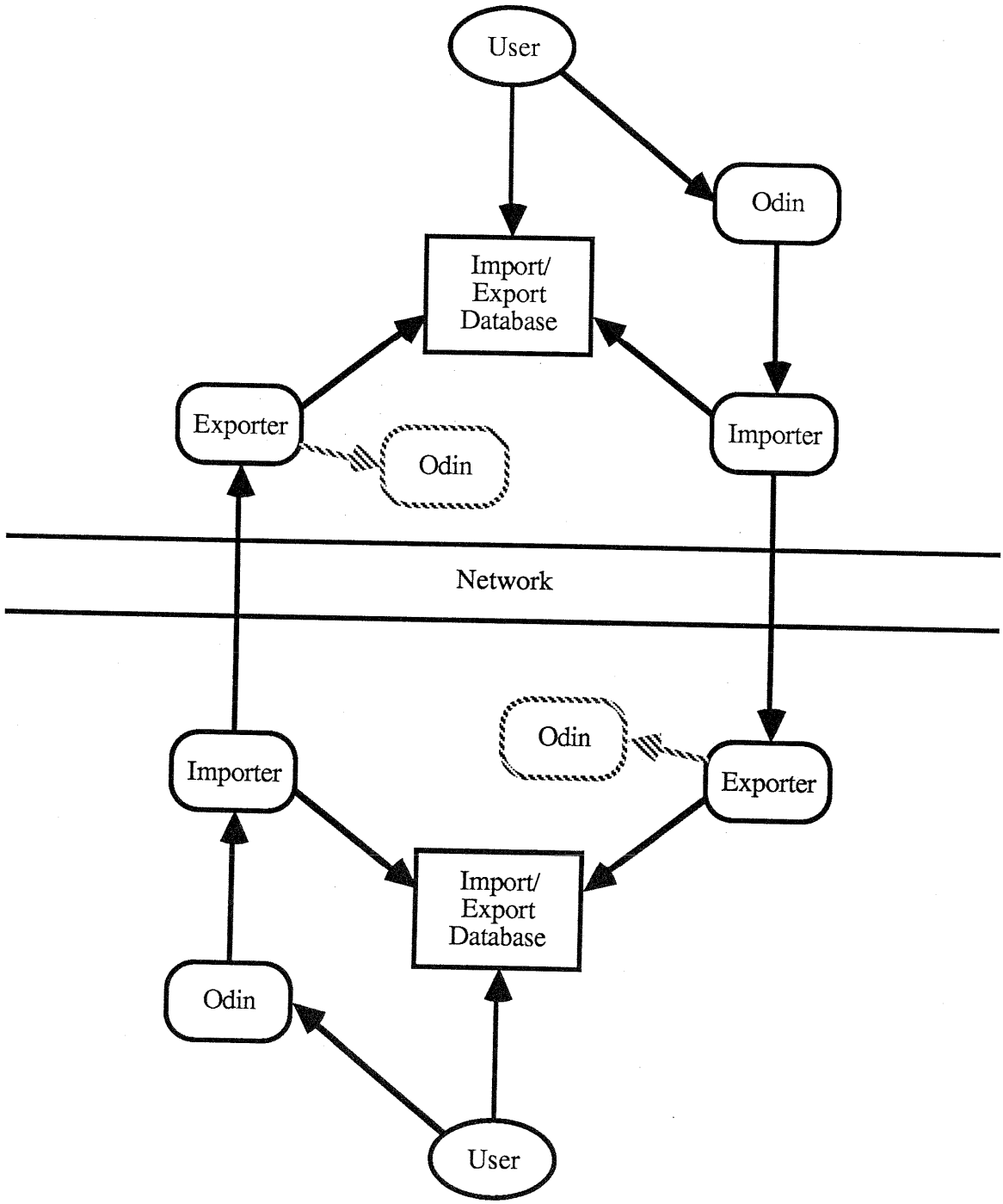


Figure 3: Keystone Processes. KEYSTONE processes for two nodes are illustrated together with intra- and inter-node control flow. The Odin process in the dashed box indicates a copy of Odin which may be invoked by the Cohort to satisfy a network request.