Data Migration in an Object–Oriented Software Development Environment

Michael H. Sokolsky
Columbia University
Technical Report CUCS–424–89

April 18, 1989
MS Thesis

Thesis Committee: Professors Gail E. Kaiser and Calton Pu

Abstract

As software systems grow from small systems developed by a handful of people to large, complex systems developed by hundreds of people, the environment in which they are developed evolves. Large software systems contain vast quantities of data that must migrate to new development environments. Object–Oriented software development environments (OOSDEs) have received research attention recently, and seem destined to become commonplace for software development of these large systems. System growth involves *data migration*, a problem that must be solved before OOSDEs become practical tools. Data migration includes *immigration* (import) of systems developed using traditional facilities, *reorganization* of systems supported by object–oriented databases (objectbases), and *schema evolution* as the class structure of the objectbase changes. This thesis presents graphics-oriented tools — Marvelizer and Organ — that solve the first two problems for the Marvel OOSDE. Marvelizer and Organ have been implemented and used on Marvel itself.

This research is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Sokolsky is supported in part by the Center for Advanced Technology.

Contents

List of Figures

1 Introduction

Software development environments (SDEs) are tools that attempt to remove much of the menial work of software development engineers, thus allowing engineers to focus on the more creative and important issues of software development. Much of the recent work on SDEs assumes *object-oriented* database support [RW89], because objects provide an excellent platform for managing the vast quantities of heterogeneously formatted data normally associated with developing and maintaining a large software system. Such environments are henceforth called OOSDEs.

Data migration of software systems involves the general process of moving, rearranging and reformatting some or all of the different parts of a software system. More specifically, *Immigration* is the process of moving a software system (or a part of one) into an OOSDE, either when starting to use an OOSDE for the continued development and maintenance of an existing software system or when reusing parts of one software system in another. *Reorganization* is the process of rearranging a software system within the framework of an OOSDE as the modular structure of the software system evolves during its lifetime. *Schema evolution* is the process of reformatting a software system as the OOSDE itself evolves, either statically as different views are employed by the users, or dynamically as the definitions of these views and their underlying object classes are revised over time.

This thesis discusses data migration in an OOSDE, and presents the implementation of immigration and reorganization tools for a specific OOSDE — *Marvel* [KFP88]. The tools are called the *Marvelizer* and *Organ*. Both tools are fully integrated into Marvel's new graphics interface, also presented in this thesis.

1.1 Immigration

The driving desire for software migration is the ability to move on to more powerful, time saving and higher quality environments. Due to short term needs, a decision to change to a new environment and face a long period without productivity is often put off, and old, cumbersome environments persevere. Adequate software immigration tools can minimize the difficulty of moving to a new environment. Often, immigration to a new environment does not result in any immediate net improvement in the software, but rather in long term improvement. Such long term gain is often difficult to justify with pressing short term needs. Most development environments available today do not support an easy immigration path to their internal format, thus immigration is a tedious exercise.

Immigration tools have a positive impact on software reuseability. If software is more accessible, it is likely to be reused, either as is, or with minor modification. Reuse is getting more and more attention from software engineering researchers today. This thesis is not concerned with reuse in an object-oriented sense, where reuse is motivated by an object's hidden structure and implementation [OHK87]. It is concerned with the accessibility of small pieces of software for incorporation into new software systems, where the small pieces might and might not change slightly from some original application.

Immigration can be either *static* or *dynamic*. Static immigration does not involve changing the structure of the system being immigrated, whereas dynamic immigration does. The Marvelizer provides a framework for static immigration of software systems to Marvel, in order to access systems which reside in other SDEs, especially those degenerate SDEs composed of collections of Unix tools. The Marvelizer does not attempt to understand any internal formats built on top of the Unix

file system. Dynamic immigration is a form of schema evolution, discussed below.

1.2 Reorganization

Continual reorganization of software is commonplace as software systems become larger. Initial views and goals of systems change, design decisions change and management changes. Often, the entire nature of a system changes during its lifecycle. Small systems often get swallowed up by larger ones, or several small systems get merged into one large one. Reorganization is often necessary to make overgrown systems more clear, space efficient, easy to debug, and easy to maintain. In most current software development environments, such reorganizations are a tedious, time consuming process.

There are very few SDEs that can generate a visual "picture" of a system, to help make it clear what reorganization is needed. Organ provides visualization through the Marvel graphics interface. Users pick objects and their new locations from a graphical picture of the objectbase.

Reorganization can be either *static* or *dynamic*. Static reorganization takes place within a particular structural framework of a system, whereas dynamic reorganization takes place when a system grows enough to make older structural methodologies inefficient. Organ is a tool for static reorganization. Dynamic reorganization is another kind of schema evolution, discussed below.

1.3 Schema Evolution

Organ and the Marvelizer are tools which in themselves are complete, but yet do not provide facilities for complete schema evolution. Such facilities include:

- External tool support, to allow activities such as parsing of files and extraction of data;
- dynamic reorganization, or schema evolution of an existing objectbase, to allow an old schema be converted to a new schema;
- and dynamic immigration, or schema evolution capabilities during immigration.

Dynamic immigration differs from dynamic reorganization in timing, however they both involve schema evolution. Dynamic immigration takes place while moving a system to a new SDE, either because the system's structure is not compatible with that of the new SDE, or because part of the immigration is to change to a more appropriate structure due to program growth or other outside factors, such as a lack of structure. Dynamic reorganization takes place more gradually, as an SDE and a system being developed in that SDE grow. This might happen, for example, when a document set is integrated with a software system. New structures and links must be developed for the SDE to manage this new part of the system.

1.4 User Interfaces

Organ in particular derives much of its power from a visual interface, where users can "see" what they are doing, rather then just doing it. In order to provide this visual interface, this thesis includes a new graphics user interface for Marvel. With the graphics interface, users have a "drawing board" in which they can perform software engineering tasks.

It is important to note that this graphics interface is specifically hand generated for Marvel, and that an exhaustive search of available graphics user interfaces was

not conducted. Furthermore, an exhaustive search of appropriate graphics interface and human factors literature was not conducted. While graphics user interfaces are unquestionably an important and interesting research area, they are not the major focus of this research.

The remainder of this thesis is organized as follows: section two provides an overview of Marvel; section three discusses why Marvel is an appropriate basis for development of immigration and reorganization tools; section four presents related work; section five describes an example which will be used throughout this thesis; the next three sections describe the design the Marvelizer, Organ and the user interfaces which support Organ and Marvel in detail; following this is a discussion of some future research areas motivated by this research; finally, conclusions are presented. Appendices containing the text of the example are included for completeness.

2 Marvel

Marvel is a knowledge-based software development environment that provides assistance in all phases of software development that contain repetitive, automatable tasks. Marvel is capable of generating a wide variety of environments tailored to support the specific needs of a wide variety of software projects. Marvel itself does not assume knowledge about any particular kinds of environments or software development methodologies, rather it is a kernel which provides facilities to generate specific environments.

First, a *superuser* writes a high level description of a software project called a *Marvel environment.* Marvel environments are written in a special-purpose object-

based language called the *Marvel Strategy Language* (MSL)[1]. Marvel environments can be written for a wide variety of things, for example, developing "C" programs, developing Pascal programs, writing books or theses, developing project management tools or even doing VLSI design.

The description of a Marvel environment can be modularized into units called *strategies*, where each strategy contains a subset of the complete Marvel environment description. MSL contains a mechanism to import and export facilities between strategies. Marvel includes facilities to dynamically *load, unload* and *merge* individual strategies to produce a target Marvel environment. Thus, one environment can contain several development scenarios to suit managers, software developers, test engineers, and so forth. Allowing different users to have different views of the environment builds in foundations of security to the system.

The two basic components of a Marvel environment are *objectbase definitions* and *rules*.

Objectbase definitions define the structure and contents of a Marvel environment. The structure is defined with *classes* and *attributes*. Classes are object groupings[2]. Attributes define the details of classes, and define a hierarchical structure for a Marvel objectbase. Attributes are *a priori* divided into *small, medium* and *large* attributes. Small attributes are simple entities such as integers, strings or enumerated values. Medium attributes are collections of small attributes, such as text or binary files. Large attributes are collections of medium attributes, such as sets of files. They are analogous to directories.

The contents of a Marvel objectbase are instantiations of the classes. We shall

[1] A syntax directed editor has been previously available to assist in writing these strategies

[2] In the literature, class and type are often used interchangeably.

refer to these instantiations simply as objects. The state of an object is defined by instantiated attributes. Objects are managed by a simplistic objectbase manager that maintains all important managerial information about each object in memory; and stores the contents, or data, associated with each object in standard Unix files and directories. We call these files and directories Marvel's *physical data space.* The objectbase manager knows how to navigate through the objectbase, create, delete and move objects about, and find the part of Marvel's physical data space that stores an object's data. The objectbase manager consults the file system only when reading or writing an actual data file. Searches and displays of the objectbase are therefore quite fast. Marvel is currently a single user system, so the objectbase manager does not handle any issues of concurrency. One of the future plans for Marvel is to incorporate a "complete" objectbase manager as part of a multi user model.

We define some additional terminology that will be used throughout this thesis. An object's *master class* is the class from which that object was instantiated. A class's attributes are called *template attributes*, to distinguish them from the attributes of an object. Objects can have *children.* Children are objects linked to their parent by large attributes of the parent. These linking attributes are called *owner attributes* from the child's viewpoint.

Marvel objects are simplistically "clustered" by maintaining an alphabetical ordering amongst a class's objects. Additionally, a second alphabetical ordering is maintained amongst an instantiated large attribute's objects. At runtime, the name of the object is used to maintain these orderings, rather then two seperate pieces of information. An extra numerical index is saved in between sessions to regenerate the second ordering.

An example clarifies this discussion. Figure 1 depicts a class structure of a "C" programming environment called *C/Marvel*. The bubbles are classes, and the bold names are large attributes that connect the classes. Smaller italicized names to the right of the classes are small attributes, that help describe the state of an object. Items in small letters are medium attributes; in this example these refer to *.c*, *.h* and *.a* files. We can see that a C/Marvel objectbase has **GROUP** objects that can have sets of **PROJECT** objects. Each project can contain a set of **LIB** objects, a set of **PROGRAM** objects, and so forth.

Marvel objects are *persistent*. Once created, they can only be removed by specifically deleting them. It is mandatory that work not be lost when a system crashes, or in the unlikely event of a Marvel crash. Persistence is achieved by maintaining an up to date state of the objectbase at all times, as well as maintaining the appropriate Unix files to which Marvel objects map. The state of the objectbase is automatically *checkpointed* after each Marvel command that changes it, therefore, old states of the objectbase can be recovered. All these previous checkpoints are easily recoverable, however Marvel leaves management of its physical file space to the tools and operating system which operate on it; thus this part can be difficult, if not impossible to restore.

The objectbase manager maintains many threads through the objectbase, in order to facilitate fast queries. See section 7.3.2 for more details.

Marvel maintains a concept of a *current object* in the objectbase. The current object is generally the object that was most recently added. Users can change the current object to be any object in the objectbase.

Rules define the activities one can do in a Marvel environment. Rules consist of *preconditions*, *activities* and *postconditions*. Preconditions of a rule narrow the

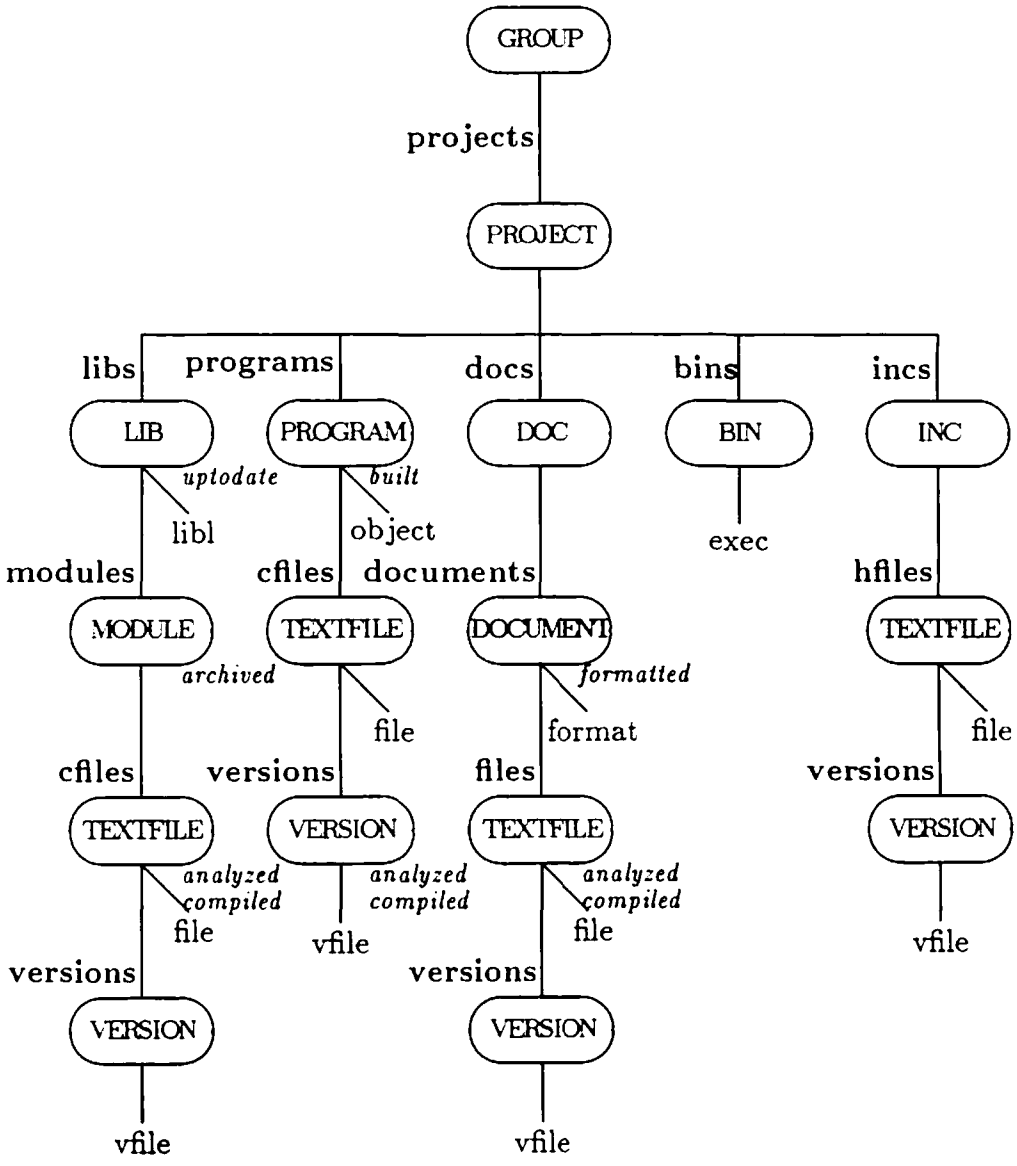

Figure 1: Class Hierarchy in *C/Marvel*

set of objects a rule focuses on by specifying a *characteristic set* of objects, and then limits this set further by specifying a set of attributes of these objects that must have some specified values. Postconditions specify values of attributes to be set. There can be multiple postconditions. Once the preconditions of a rule are all satisfied (true), the activity part of the rule *fires*. Firing means executing the activity part. This is done with an *envelope*. An envelope is a COTS[3] tool or Unix shell script. Postconditions are asserted based upon the results of the envelope. Thus envelopes act as a liaison between the Marvel objectbase and the Unix file system.

The Marvel kernel is a *controlled automation* engine. It uses the preconditions and postconditions of rules to form a network of possible automatic multi-rule activities. Once an initial rule is manually invoked, other rules will be invoked as needed to satisfy the original rule's preconditions. This is called *backward chaining*. Backward chaining is essentially what the *make* [Fel79] family of tools does. After the target rule is finished, other rules are invoked if their preconditions have since been satisfied by the assertion of the original rule's postconditions. This is called *forward chaining*. Backward and forward chaining continues *opportunistically* (as the opportunity arises) until there are no more possible chains.

Marvel has a menu based, mouse driven graphical interface. This interface is presented in detail in section 8.2. A command line driven line oriented interface is also maintained for those not fortunate enough to possess a graphics workstation. All the facilities of Marvel are available from either interface. Section 8.3 and [BS88] describes the line interface.

[3]Commercial Off the Shelf

3 Background Research

This section discusses the background research that led to the conception of Organ and the Marvelizer. Included is a discussion of why Marvel was chosen as a platform for implementation of Organ and the Marvelizer. Several other systems are mentioned which provide the beginnings of the facilities provided by Organ and the Marvelizer.

3.1 Why Organ and the Marvelizer?

The idea of visual databases is not new [BL86], but its practical application to SDEs is. This comes as a surprise, considering the state of graphics systems. However, the surprise is diminished when one considers the state of SDEs. A major problem with SDEs is the commitment to the tools they provide. The potential power of toolsets such as Unix is taken away with the commitment to an SDE. There have been several SDEs which provide a paradigm for programming [Sun88,SKHA86], yet it is unclear whether these environments scale up to the task of large, evolving systems. Evolution appears as one of the least considered elements of these environments. For a system to evolve, reorganization and migration are needed.

3.2 Why Marvel?

Most software development environments provide a "canned" environment for a certain task, for example, SMILE [KF87] is a "C" programming environment. However, development methodologies and requirements for software vary widely from one organization to the next, so these "canned" environments often compromise

power in exchange for convenience. This is unacceptable. This might happen, for example, in a system that contains a mixture of source code languages and documentation strategies, or in a system where management imposes stringent controls on development. Many organizations choose to create and recreate their own environment in parallel with their system's development, rather then use a commercially available[4] software development environment. This leads to great wastes of redevelopment time. Marvel provides facilities for the parallel development of a system and an environment with reduced environment development and modification time.

Architects of software development environments often create their own tools for certain familiar tasks, such as automatic program compilation. This is somewhat problematic, because it forces the user who might be familiar with a common tool to learn something new. Many of these new tools do one specific part of some older tool very well, but are not as complete. As an example, consider Unix's `make` [Fel79] tool. Make is not especially sophisticated, in that it knows nothing about the process of any kind of software development. Yet numerous similar tools have been developed that try to perform `make`'s task better, such as DSEE [LRPC84] and Space [ML88], but are only questionably successful.

With Marvel, the development time to develop a "custom" system is reduced to the time it takes to modify a set of strategies and envelopes to match local design methodologies. Initial indications are that this task is much less difficult than starting from scratch. As Marvel is used more, the base of strategies and envelopes available should be large enough that this task becomes very simple.

The Marvel kernel provides a methodology for using standard system tools, rather then starting from scratch. Furthermore, Marvel environment's can be tai-

[4]Or semi–commercially available, at least.

lored to match the needs of different individuals and groups. Additionally, sets of strategies can be developed to allow arbitrary side by side development of program source code, documents, or any other large body of work which requires repetitive, menial tasks to be performed during it's development.

Thus Marvel provides many of the facilities needed to be an effective SDE. While immigration and reorganization tools still do not necessarily make Marvel "complete", they make it a practical environment for software development. Marvel's architecture is open, as discussed above, so reorganization and immigration tools can be general, and easily applied to other SDEs or objectbases.

4 Related Work

I know of no SDE's, object–oriented or otherwise, that have acceptable facilities for immigration and reorganization, as described in this thesis. Many environments incorporate some of the ideas of this thesis. Following is a description of the relevant features of of these environments. I have tried to encompass much of what exists in this area in the following several examples.

Most SDEs ignore the fact that a developing system will undoubtedly have to undergo major reorganization before it appears in final form. Consider Unix, which many software developers use instead of an SDE. Most Unix facilities for reorganization consist of operating system level commands for organizing projects. Relevant Unix commands include `mv`, `cp`, `ln`, `ls`, `tar` and `cpio` [Sun86]. Unix provides very little visual assistance in reorganization tasks. On a local site by site basis, Unix shell scripts that utilize these commands are often written to help perform basic reorganization of software projects. These scripts tend to be unsupported

and have errors, because developers would rather be doing "real" work, instead of taking time out to work on software development tools. As *ad hoc* environments develop, more and more time is invested in writing "custom" scripts and programs to suit a system's needs. Such "custom" tools are often recreated from system to system. Rarely are any of these tools visual in any way.

SMILE [KF87] has a concept of experimental databases, which are conceptual "copies" of an entire system, in which specifically reserved objects can be updated. They can be used as entities for reorganization by appropriate manipulation of experimental databases, and by a retrieval feature which allows incorporation of a module from another SMILE database. However, in SMILE this is very cumbersome. SMILE does not have any specific facilities for conversion to their format (which is far from trivial by hand), thus in SMILE it is difficult to reuse components from older systems, or immigrate systems already under development.

NSE has conceptually similar experimental databases to those of SMILE which they call private workspaces. They differ from SMILE's in that they are based much more on the native Unix file system (NSE's base operating system), and thus are much easier to reorganize. Any currently existing UNIX command or tool can interact with NSE, in addition to the private workspace manipulation provided by NSE. However, NSE maintains internal system specific information that might be invalidated by stand alone Unix commands such as `cp` (copy) reorganizing directories everywhere. It is much less cumbersome to immigrate a system to NSE than to SMILE, allowing for easier reuse of old code. This is accomplished with a tool NSE calls `bootstrap`. In practice, `bootstrap` is a complicated process because NSE specific Makefiles and component description files must have been previously generated.

Infuse [PK87,KP87], a change management tool, also has an indirect, but potentially more powerful notion of reorganization then SMILE or NSE. Infuse partitions sets of modules that are being changed into a hierarchy of experimental databases. These hierarchies can be repartitioned as more appropriate partitions are found. Thus, Infuse generalizes SMILE's experimental database concept. This could conceivably be the basis for reorganization based on similarity. Infuse could be combined with a system such as Marvel to gain a more complete software development environment. It is unclear how easy this would be.

Software Project Management System (SPMS) [Hew86] provides an automatic framework for doing hierarchical file system oriented operations based on a set of assignable attributes, and simple logical combinations of these attributes. Actual operations on a system, for example compiling, releasing or reorganizing an attributed set of directories in a particular way, must be provided by the user, who must often fall back on ad hoc scripts and programs.

The few object–oriented databases available commercially, such as *Vbase* [AH87] and *GemStone* [PS87], do not yet provide specific data migration facilities for reorganization or immigration. Skarra and Zdonik discuss schema evolution in an OODB in [SZ86], and present some possible solutions. Schema evolution in OODB's today is unquestionably a universally recognized problem, however, we know of no systems that support the entire data migration problem.

Some operating systems and SDEs provide simple visual extensions to tools to make them visual-oriented. A hierarchical version of `ls -R` (a recursive file lister for Unix) that tries to show the hierarchy more clearly is a good example. Many systems have such a tool. There is much less available when it comes to viewing objectbases. A visual tool for managing relational databases is presented in [BL86].

There are many tools that support some form of program structure visualization. PECAN [Rei84] is an example of a tool that visualizes program execution, in a nice graphics based programming environment.

5 An Example: *C/Marvel*

Throughout this thesis, examples will all refer to a Marvel environment called *C/Marvel*, developed by the author of this thesis and others [BK88]. *C/Marvel* is an implementation of a C programming environment for Marvel.

Figure 1 contains a description of the class organization of *C/Marvel*'s objectbase, for use as a reference in the discussions of class structure in this thesis. *C/Marvel*'s class structure was derived by looking at the organization of the Marvel software itself. As a testimony to *C/Marvel*'s functionality, Marvel can currently be developed with *C/Marvel* in a single user mode.

Appendix A contains the MSL objectbase definitions used in *C/Marvel*, and Appendix B contains the full set of rules used in *C/Marvel*. In practice, these rules would be broken apart into several separately accessible strategies. This way, the Marvel environment could be configured differently for different kinds of users (such as managers and software developers) depending on individual needs. Such configurations enforce controlled use of tools not appropriate for all users.

C/Marvel includes many important facilities C programmers require to effectively develop C programs. These facilities include:

- compiling files, with `make` and `cc`,
- analyzing files, with `lint`,

- library maintenance of modules, with `ar`,
- building programs, with `ld (cc)`,
- editing text files, with a user's favorite text editor,
- viewing text files, with `less`,
- debugging, with `dbx`,
- revision control, with `RCS`,
- and executing programs.

For more information on *C/Marvel*, see [BS88].

6 Design: The Marvelizer

We have defined the immigration of a software system to be the process that includes moving all the pieces of that system to a new or updated OOSDE. Most OOSDEs assume that systems under development find their own way of immigrating from one environment to the next. This section describes the Marvelizer, an automated tool for static immigration into Marvel. First, a discussion of some of issues involved in software migration is presented. Next, the design goals of the Marvelizer are presented. Then, pertinent details on the Marvelizer's implementation are presented. Finally, a discussion of the additional facilities the Marvelizer needs to be a truly effective tool for dynamic immigration, or schema evolution, is presented.

6.1 The Immigration of a Software System

Most SDEs assume that systems under development will just find their own way of migrating from one environment to the next, or from one development platform to the next. Few systems, a notable exception being Sun's NSE [Sun88], provide any formal facilities for the immigration procedure. For most software development environments, the immigration procedure is manual and very complex. Most environments have their own sets of tools, each of which require specifically formatted data. Furthermore, environments themselves often maintain their own data structures to represent the actual body of the system. These data structures must often be initially derived with awkward, ad hoc methods.

Most OOSDEs have their own sets of tools, each of which require specifically formatted data. Furthermore, environments often maintain private data structures to represent the entire system. These data structures are often difficult to derive. Like other SDE's, Marvel maintains an internal objectbase definition. However, the bulk of the real data resides on the file system in a relatively standard hierarchical tree of files. Mappings to this structure, and the current values of the objects' small attributes, are all that are maintained internally by the objectbase manager. Nonetheless, this objectbase is not easy to hand generate, as it resides in a very strict format computer generated file. Directory structures can be manually moved into Marvel, however the user must be more familiar with the objectbase structure then is otherwise required to effectively use Marvel.

Immigration is made more complicated because there is often little in common from one organization's methodology of software development to another's. Designers and implementors of SDEs certainly can not be expected to support a large collection of system organizations in their immigration procedures. All but specific

target customers tend to have difficulty immigrating from one system to another. because SDEs tend to support some specific set of tools and software organization in an inflexible manner. In Marvel, there are no dedicated environment specific tools; Marvel can support most COTS (Commercial Off the Shelf) tools and locally developed and already in use tools. Furthermore, Marvel allows very flexible objectbase class structures, which can match to most hierarchical system organizations.

6.2 Design Requirements

The central requirement of the Marvelizer is to immigrate a software system into Marvel automatically, without the user being subjected to the creation of complex mappings or data files. Furthermore, the immigration must take place in a timely fashion. Timely here means on the order of how long it takes to move the same software system to a different place in the file system with a standard operating system command. A command such as `cp -r` for recursive directory copying, is a timely example in the Unix domain.

The Marvelizer must generate an appropriate Marvel objectbase, complete with its physical file structures. It must handle entire systems, where it creates a new objectbase; and chunks of systems, where it appends to an existing objectbase. Handling chunks is mandatory for reuse, slight modification or incremental Marvelization of a system.

Easy visual verification of the results of immigration is necessary. Since very large quantities of data can potentially be immigrated, it is important to be able to verify that correctness of the immigration without an excessive expense in a user's time.

Marvel keeps its data in a relatively standard hierarchical tree of files, but manually moving a large body of code into Marvel's format would be oppressive, and difficult to do without omissions and mistakes. Even if this migration were performed by hand, Marvel's parallel internal objectbase structure would not exist. While this information is kept in a readable file, and is editable[5] with a normal text editor, this is not a reasonable expectation from any group migrating to a new, supposedly better and more automatic SDE.

The Marvelizer must be able to run in an interactive mode, so that if it has not been given enough information about certain choices, it can query for appropriate answers. However, the information the Marvelizer needs from a user must be kept simple and concise. An unreasonable requirement from a migration facility is one which requires specific, difficult to generate input files or maps which in some way "describe" the input being migrated.

6.3 Implementation

Prior to Marvelizing, a "Marvel Administrator" must create a Marvel objectbase structure which realizes the directory structure of the software system to be Marvelized. This objectbase structure must not deviate much from the structure of the existing system. We do not expect average users to be Marvel administrators. An example of such a structure has been presented in figure 1.

The Marvelizer maintains a table of user supplied immigration rules. These rules are consulted before the immigration of each object. Rules are one line entries into a table the Marvelizer maintains. There are two kinds of rules the Marvelizer

[5] But this file is very cryptic, and only a Marvel guru is likely to edit this file with a chance of making no grave mistakes.

```
Now enter all medium attribute suffixes, for each class.
Only one class per line, please.  Format is:

CLASS_NAME <suffix-1> <suffix-2> ... <suffix-n>

Enter a q when finished, or an e to exit.
Enter string: LIB .a
Enter string: TEXTFILE .c Makefile .mss .tex .bib .y .lex .h
Enter string: VERSION ,v
Enter string: BIN marvel loader reserve deposit
Enter string: q
```

Figure 2: Suffixes needed to Marvelize Marvel.

uses, as follows.

- First are rules which provide the Marvelizer with information about the kinds of files it must handle during Marvelization. These rules specify the suffixes of all the different file types each class might contain. Suffixes can match entire file names. These rules appear as follows:

  ```
  <class-1> <suffix-1> ... <suffix-n>
  ...
  <class-m> <suffix-1> ... <suffix-n>
  ```

  where $m \geq 0$ and $n \geq 1$.

  Figure 2 shows the suffixes needed for a complete Marvelization of Marvel. The format of this input is discussed shortly. The input displayed in figure 2 is used for the Marvelization in figure 5.

- Second are rules which specify directories that belong to particular classes. These rules can minimize interaction with the Marvelizer, and provide a basis for future batch capability. These rules appear as follows:

```
<class-1> <directory-name-1> ... <directory-name-n>
...
<class-m> <directory-name-1> ... <directory-name-n>
```

where $m \geq 0$ and $n \geq 1$.

Several different levels of verbosity and error recovery are supported, as follows.

1. Verbose mode. The user is constantly supplied with messages saying what the Marvelizer is doing. Furthermore. if there are questions of where to place a directory, the user is queried for either the correct large attribute of some object, or to skip the directory.

2. Quiet mode. This mode is similar to verbose mode, however the user is not supplied with the constant messages saying what the Marvelizer is doing. The query facility is still there.

3. Silent mode. In silent mode, if there are questions about where a directory belongs, it is quietly skipped. While this in not an optimal policy, it allows for a skipped directory to be later Marvelized without first having to cleanup improper actions on the Marvelizer's part. This mode is useful if a user is sure of the success of something, and does not want to be present to answer queries. This mode is provided to allow future batch Marvelizations; it currently allows a methodology for automatic Marvelization.

A dialogue with the Marvelizer is presented in figure 3. This dialogue is from a Marvelization of Marvel itself, using the C/Marvel environment mentioned earlier. User responses are in italics. Figure 4 shows a mostly empty objectbase. Figure 5 shows the final results of the Marvelization of Marvel, as described in figure 3.

```
Enter the file system root to be marvelized:  /example/marvel
Enter target class for /example/marvel:  PROJECT
(v)erbose, (q)uiet or (s)ilent mode?  (any other key to exit):v


Now enter all medium attribute suffixes, for each class.
Only one class per line, please.  Format is:

CLASS_NAME <suffix-1> <suffix-2> ... <suffix-n>

Enter a q when finished, or an e to exit.

Enter string:  FILE .c .o Makefile
Enter string:  VERSION ,v
Enter string:  q


Now enter multiple classes and objects for any given level of hierarchy.
Only one class per line, please.  Format is:

MULTIPLE_CLASS_NAME <object-1> <object-3> ... <object-n>

Enter a q when finished, or an e to exit.

Enter string:  PROGRAM marvel loader
Enter string:  q
Ready to marvelize /example/marvel.  Are you sure [y/n]:  y

    ...
```

Figure 3: A dialogue with the Marvelizer.

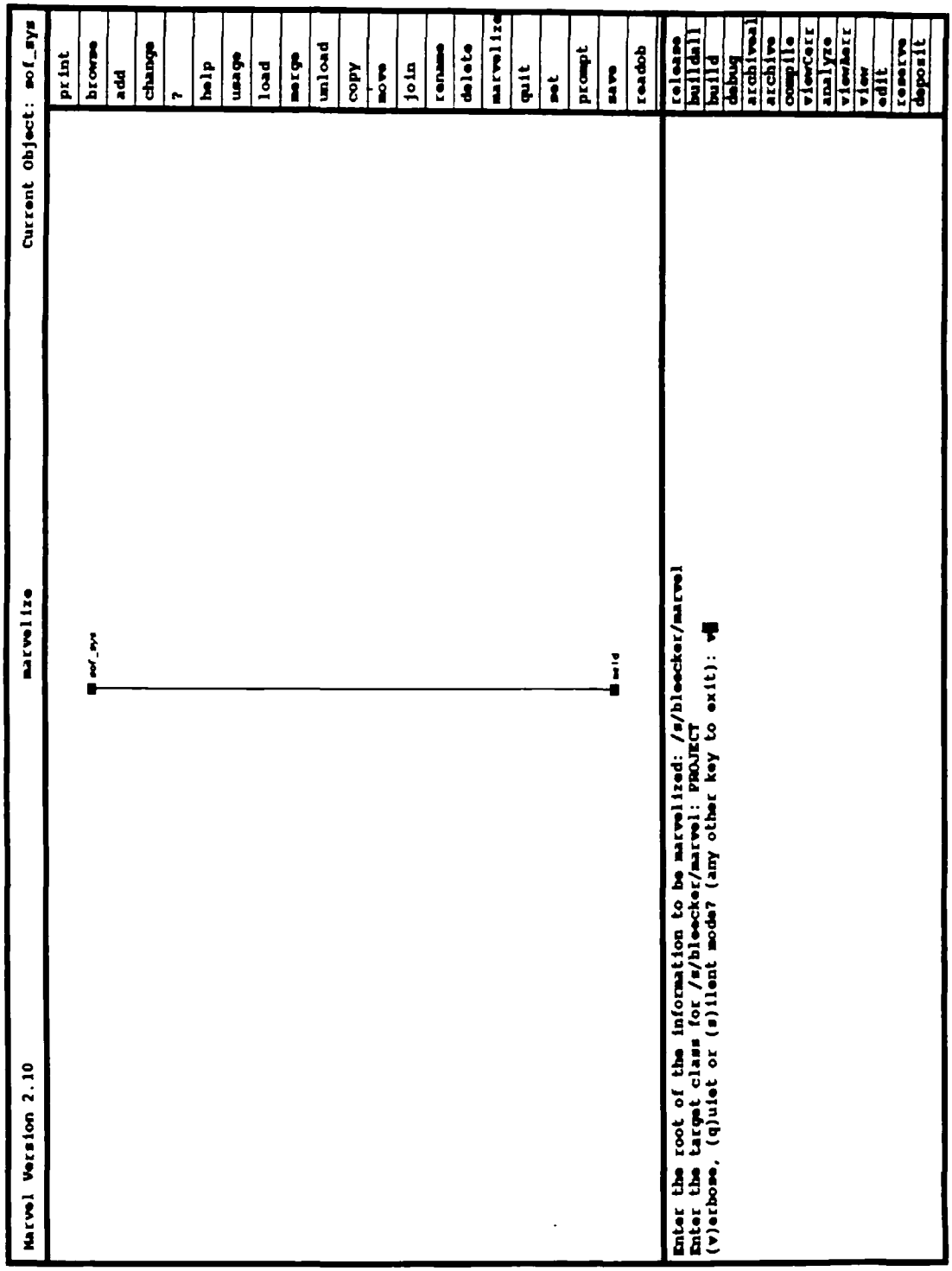

Figure 4: A Marvel objectbase before Marvelizing.

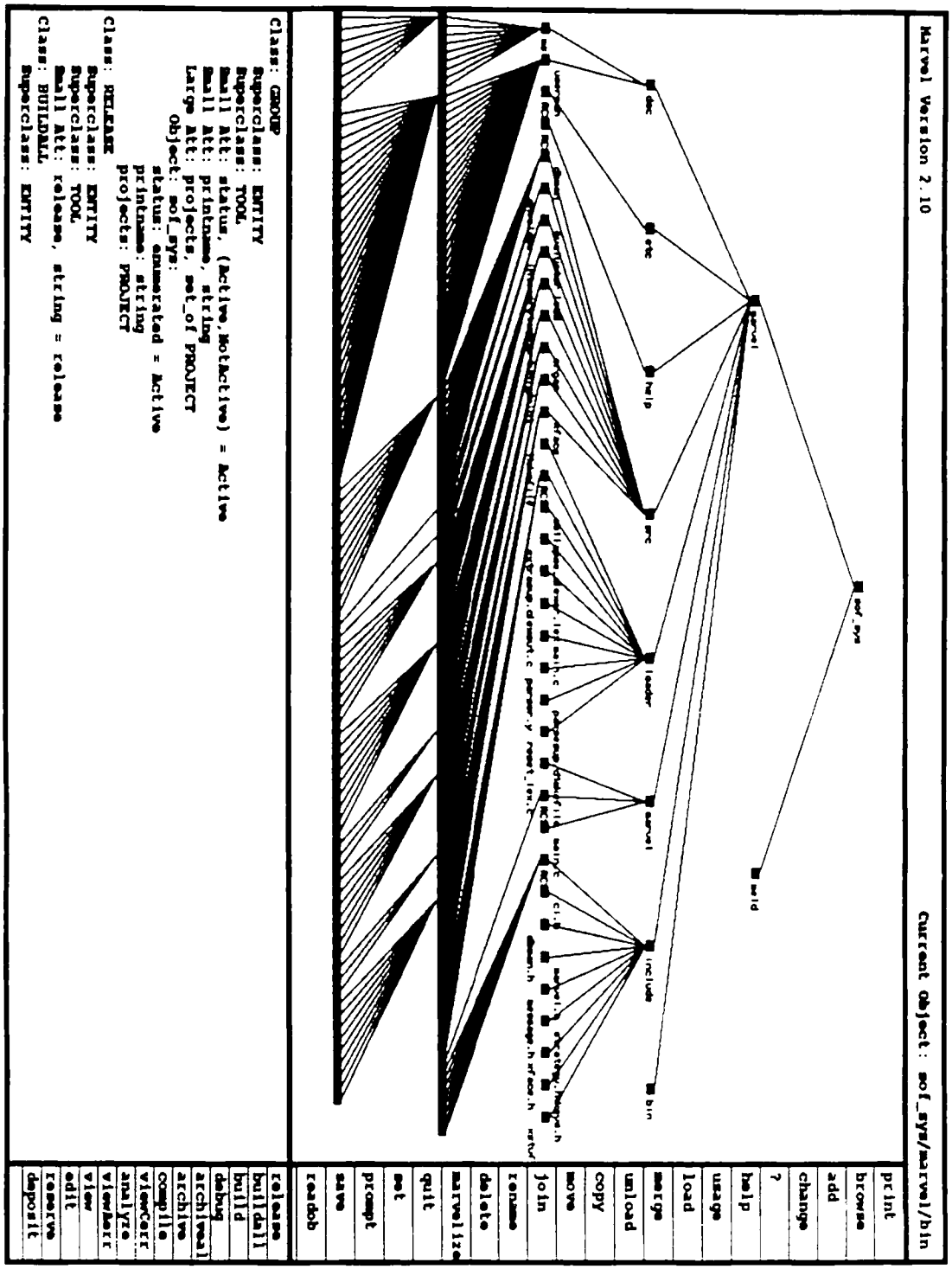

Figure 5: The same objectbase after Marvelizing.

The Marvelizer uses the Unix C library functions for reading directories[6] to do a preorder traversal of the directory tree being immigrated. A parallel preorder traversal of the Marvel objectbase is done, starting at the root class specified. Marvelization is accomplished in a single, read only pass of this directory tree.

The root object is treated specially. It has been verified during the initial query stage, so the Marvelizer immediately adds it using the add command (discussed later). This new object is set to be the system's current object. Then the traversal of the directory being immigrated begins.

When a file is encountered, the Marvelizer looks to see if the file's suffix matches a rule in the table, and if that rule's class belongs to either the current class, or a large attribute of the current class. In the first case, the file is simply copied into an appropriate place in Marvel's physical data space. In the second case, a child object in the rule's class is hierarchically added, deriving its owner attribute from the current class' large attributes. Then the file is copied to that child object. Refer to figures 1 and 3 for an example. If the current class is **MODULE**, and a `.c` file is encountered, an object of class **TEXTFILE** is added, using the *cfiles* large attribute of the current object as an owner attribute. Then the `.c` file is copied to the new object. Files not found in the rules are ignored. Messages specifying the ignored files are generated, depending upon the verbosity level.

When a directory is encountered, the Marvelizer first looks to see if there is a rule specifying it. If there is such a rule, and if the rule's class is a member of the current class' list of collection attributes, then a hierarchical object is added, as described in the second case above. Otherwise, the Marvelizer determines the set of possible classes this directory could be an instantiation of, based on the current

[6] `opendir()` and `readdir()`.

class' template attributes. If there is more than one element in this set, the user is queried about possible class choices. The user picks a class (possibly ignoring the directory is an option) and the object is added as above. The system's current object is moved (hierarchically down) to the newly added object, and the Marvelizer creates an appropriate directory in its directory structures. Recursive processing continues.

The display is updated after Marvelization. At this point, any problem directories can be Marvelized again, using different starting points in the Unix file system and the objectbase. A problem directory has a structure that does not match Marvel's objectbase definition. The Marvelizer skips such directories. The program structure can be changed in this fashion.

6.4 Schema Evolution

Implementation of the Marvelizer has shown that migration of software is a difficult task. The most serious limitation of the Marvelizer is the lack of ability to comprehensively restructure a software system on the fly. We refer to such restructuring as *dynamic immigration*, or schema evolution.

During the implementation of the Marvelizer, several ideas for dynamic immigration in Marvel have come up. These ideas are discussed in section 9.4.

7 Design: Organ

This section describes the core of the Organ system. First, initial design requirements are presented. Second, each command is presented in detail. Third,

some interesting details of the Organ command set implementation are presented. Finally, an examination of the necessity and completeness of the Organ command set is presented.

7.1 Design Requirements of Organ

The basis of the Organ command set is the command set offered by Unix for reorganizing file systems. The relevant Unix commands include `mv`, `cp`, `ln`, `ls`, `tar` and `cpio`. Unix provides very little visual assistance in reorganization tasks. The basic premise of the Organ command set is to provide a more powerful set of commands then Unix offers specifically for reorganization of software systems, with complete visual assistance to users who have an appropriate graphics display. For users on a standard terminal, the system is intended to have full capabilities, but without the assistance of the objectbase display facility and graphics browser.

Organ commands all assure internal consistency of the in–memory objectbase upon completion. All routes to failure are examined prior to any actual manipulation of the objectbase. The objectbase is checkpointed immediately prior to the start of any Organ command, and upon completion of each command. Commands do not modify Marvel's physical data space until the in–memory objectbase is successfully updated. Thus, an Organ command is an "atomic" transaction, where recovery from a system crash or network failure simply involves the restoration of the previous state of the objectbase, which is kept on disk in a readable text file. Organ does not currently address operating system or network failures that can corrupt Marvel's physical data space. Success is otherwise assured, because of the successful completion of the manipulation of the in–memory objectbase.

Organ commands are menu and mouse driven when using the graphical interface. Users are constantly informed of a command's status, and when a command is waiting for input.

The text based versions of the Organ commands are similar, except that all objects must be chosen by specifying qualified paths. In practice, the graphics display seems to be a great aid to the reorganization task.

The time limiting components of the Organ command set must all be the actual operating system time involved in carrying out the physical disk activity specified. Thus, the objectbase design must be efficient enough to allow searches for objects in O(c) time, rather then searches down potentially long lists of objects.

Organ commands are not fragile. As appropriate, they resolve duplicate name conflicts on the fly, warning users as need be. In general, if it is less work to rename an object after an Organ command, Organ will generate an appropriate, unique name. Otherwise, Organ will fail, and the command must be reexecuted. This name resolution methodology avoids repeating more complex commands.

However, Organ commands do not assume intelligence about what the user had in mind. Thus, if two objects are chosen whose relationship is unclear, Organ will specify the problem as accurately as possible, and return control to the user without invalidating the objectbase. The user can then reissue the command appropriately.

7.2 The Organ Commands

Organ commands allow users to add objects , copy objects (and all their children) to other objects, move objects (and all their children) to other objects, join the children of two objects, rename an object, and delete an object (and all of its

children).

In the following discussion, the *source* object is acted upon and the *target* object receives the activities of the command. Commands which operate on two objects do so by first specifying a source, and then specifying a target. Those which operate on just one object specify only the target.

Organ commands all have a consistent user interface as follows:

1. Pick the desired Organ command from the menu.
2. Pick an object in the `display` window after the prompt.
3. If the command operates on a source and a target, pick a target in the `display` window after the prompt.
4. Wait for the command to complete and the display to refresh itself.
5. Either pick **Done**, or repeat the above steps.

7.2.1 Add

`Add` creates an object in the Marvel objectbase. It is a fundamental command, all other Organ commands are built upon this basic facility. `Add` must be used when initializing an objectbase. It can then can be used to add children to that object, according to the currently defined class structure of the objectbase.

`Add` supports the addition of new objects *horizontally* and *hierarchically*. Horizontal addition adds the new object to the current class. The new object becomes the current object. Hierarchical addition adds a child object to the current object using one of the current class's large template attributes as an owner attribute. This

large attribute need not be instantiated in the target yet. A link from the current object through the specified attribute to the new object is created.

The user interface to add is an anomaly amongst the Organ commands. While it is concievable to point to names, classes or parent objects to provide add with information, it is impossible to always provide enough information in this fashion. Instead, the user is queried for directions on doing a horizontal or hierarchical add, and then the user supplies an object name and (for a hierarchical add) an attribute name. Instances are added relative to the current object, which eliminates the need to pick objects on the display. Only one object can be added with each invocation of add.

As an example, consider an empty objectbase in the *C/Marvel* environment. The following commands create a top level **GROUP** object called *software-systems*, with three children, all of whom are **PROJECT** objects. The commands are shown with their line oriented interface flags for clarity. Figure 6 shows what the objectbase looks like after execution of these commands.

```
add software-systems
add -a projects marvel
add -a projects meld
add -a projects mercury
```

7.2.2 Copy

Copy nondestructively copies a source and all of its children to a target. The target's master class must have a large template attribute that the source can use as a linking owner attribute. This large attribute need not be instantiated in the

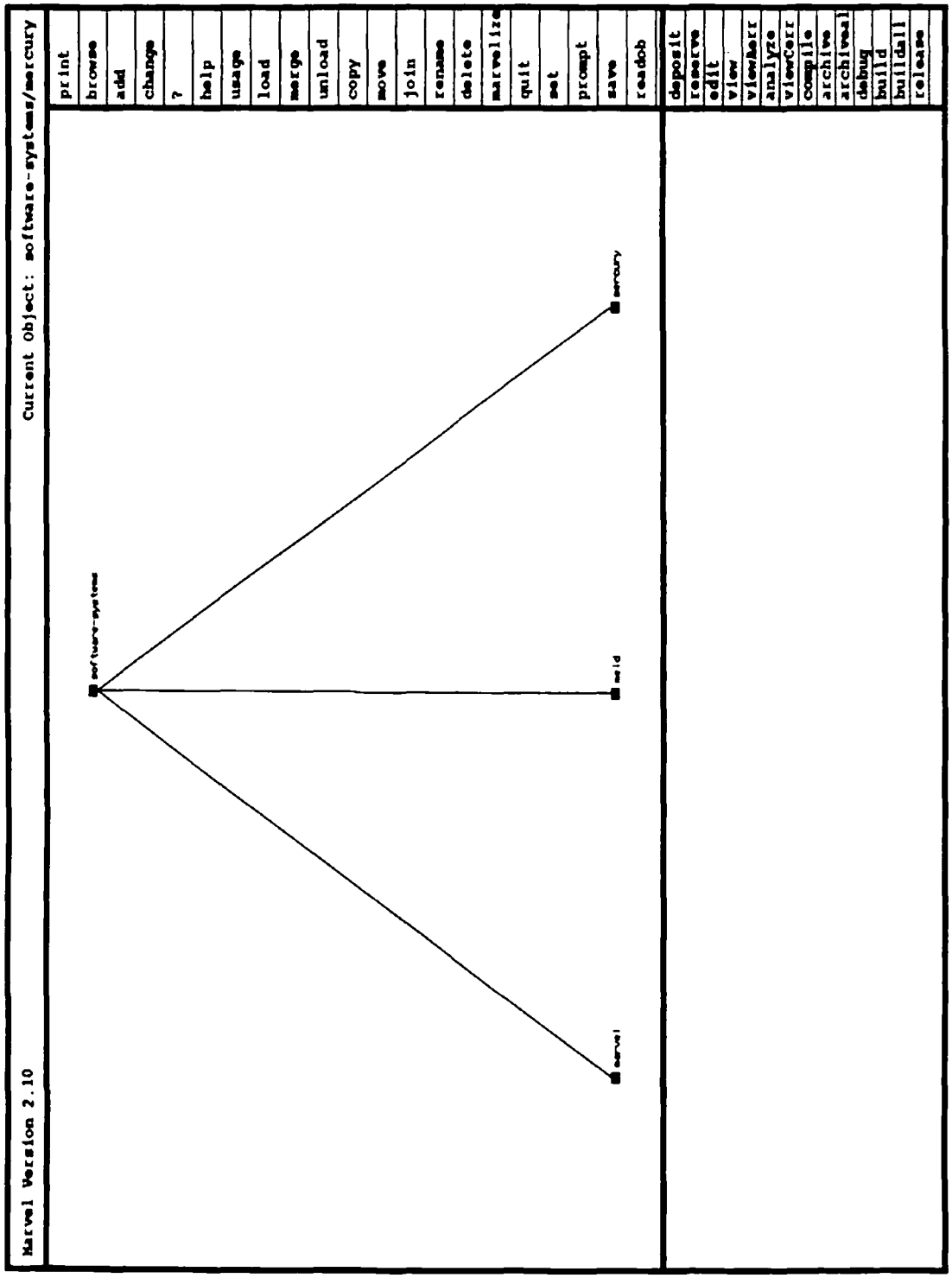

Figure 6: An objectbase after four add commands.

target object yet. This restriction prohibits coercing objects of one type into objects of another type. Such coercion may be desirable, and is discussed later. An object is not allowed to be copied to itself.

Copy repeatedly calls add to create all the new objects below the target from the source. Marvel's physical data space is then updated. The objectbase manager supplies appropriate mappings to files and directories to do this. Copy has no difficulty with end cases such as an entire objectbase or a simple leaf object.

7.2.3 Move

Move destructively moves a source and all of its children to a target. The user interface and restrictions for move are identical to those for copy.

The difference lies in the actual work move does. Move simply unlinks the source and relinks it in the target. Recursive moving of the source's children is automatic, because the links of Marvel's objectbase extend to the next level of hierarchy down or up.

7.2.4 Join

Join destructively joins two objects. To be joined, both the source and the target must be members of the same class. The source disappears, and all of its children become part of the tree rooted in the target's parent. Join is undefined for top level objects. It should be noticed that Join is fundamentally different from a relational database join operator.

Join is a move of each of the source's children to the target, and is implemented as such. An example from the *C/Marvel* environment shows the power of join.

Joining two instances of the class **LIB** could potentially involve moving many, many modules, because libraries are often vast. This is too time consuming without a command such as `join`.

Figure 7 shows an object called *src*, belonging to class **LIB**. A `join` command has just started, and the object *Organ*, belonging to class **MODULE** has been picked as the source. Next, *xface* is picked (also belonging to class **MODULE**) as a target. Figure 8 shows the results of the `join` command.

7.2.5 Rename

`Rename` renames a target object. In order to assure (alphabetical) order in the objectbase, a renamed object must be completely unlinked from its old location, and relinked into its new position, relative to its new name. Maintenance of order is necessary to guarantee fast objectbase access. The actual object is never destroyed. This way, all of the renamed object's children are moved automatically with the object. Since many components of an objectbase might change location relative to each other with one `rename`, the visual effect of `rename` is often larger then one might expect. Marvel's physical data space is unchanged except fro a rename of a directory. The operating system maintains its own data structures for locating directories and files.

7.2.6 Delete

`Delete` deletes a target object and all of its children from the objectbase. `Delete` can potentially remove an entire objectbase in this fashion. Due to the severity of `delete`'s actions, the user is asked to be certain of the deletion.

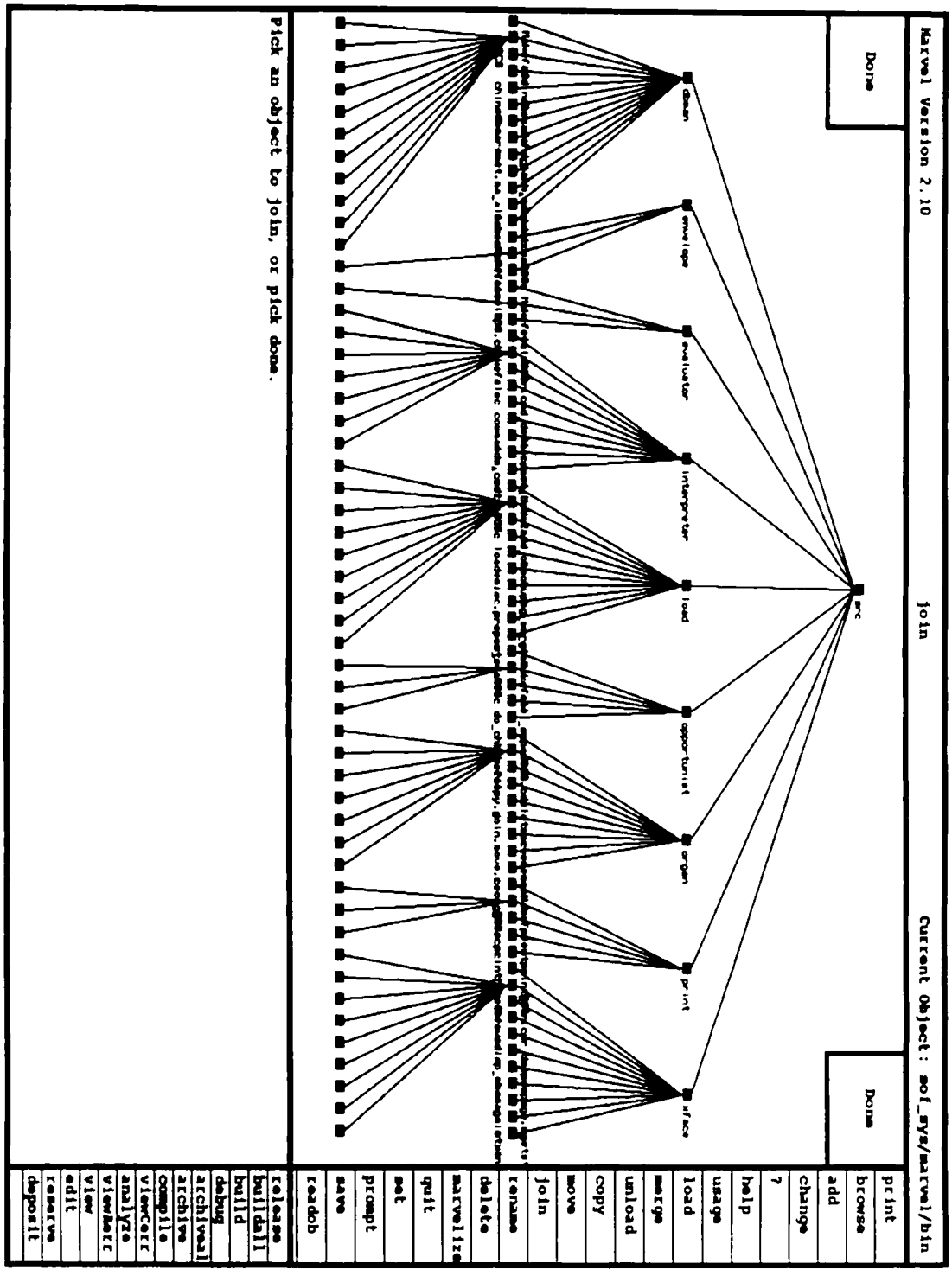

Figure 7: A Marvel objectbase before join.

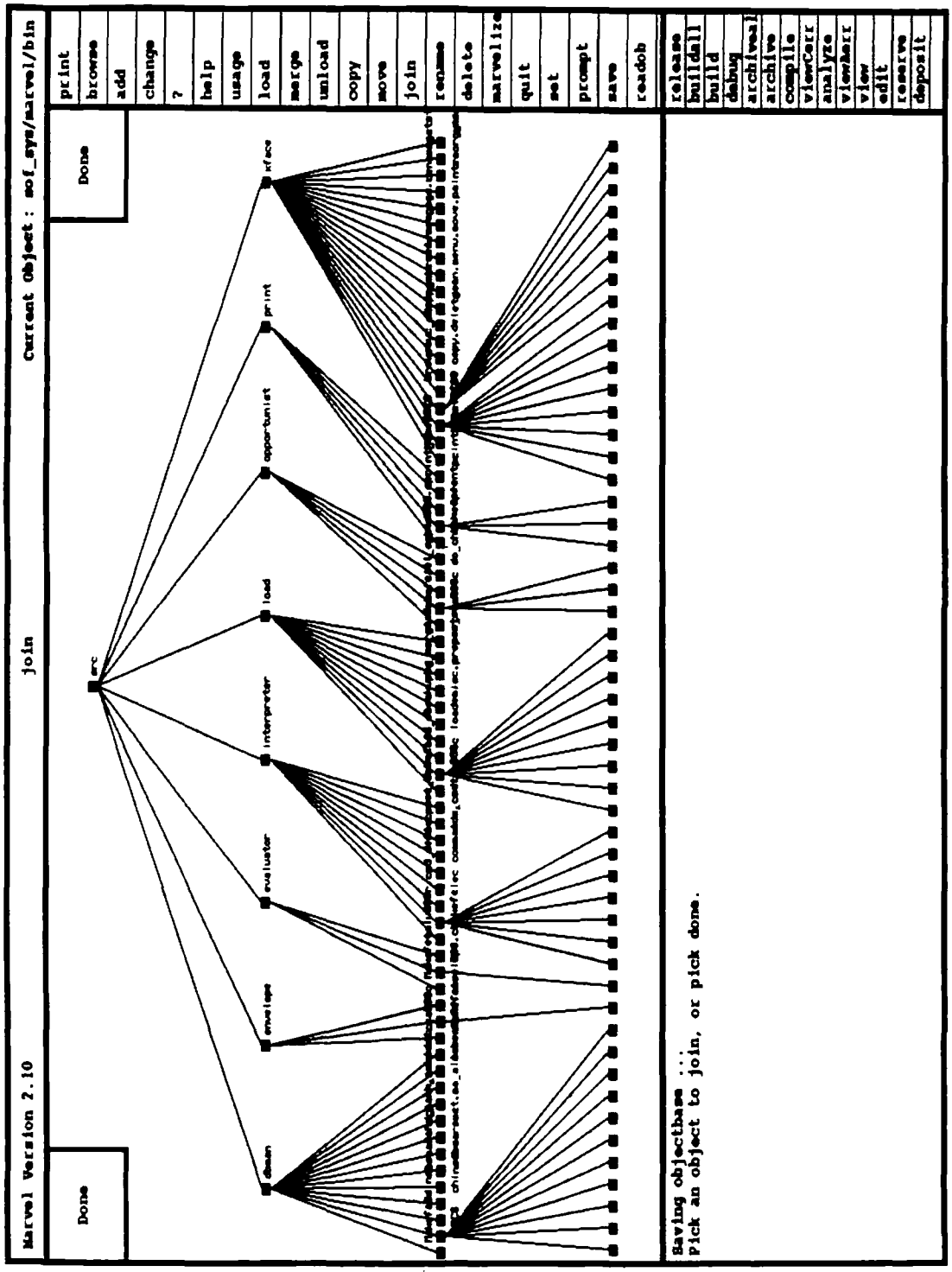

Figure 8: The same objectbase after the join.

7.3 Implementation of Organ Commands

The previous sections described the overall operation of each Organ command. This section focuses on the remaining details of implementation appropriate to this thesis. Further details can be found in [Sok89].

7.3.1 Naming Conflicts

In all commands except add and rename, name conflicts are handled without failure. Upon discovery of a name conflict, a unique name is generated for the source. The unique name has the same root as the original name, hence it is easy to pick the object out and rename. A warning is issued when Organ must generate a unique name. Targets are never renamed. Organ's name conflict resolution policy is to automatically resolve name conflicts with a unique name (rather then exiting in error) when it is less work for a user to rename an object than to repeat the command with some other name. It is unclear that this is the best name resolution policy.

7.3.2 Speed Analysis

The objectbase manager maintains links and tables in the objectbase to allow O(c) graphics search, insert and delete time and O(n) non-graphics search, insert and delete time. c represents the constant few simple calculations needed to translate a pair of graphics coordinates into indices of an object lookup table maintained as part of the objectbase. n represents the number of objects in the objectbase. The non-graphics search time is slower because the object must be specified by a *fully qualified Marvel path.* A fully qualified Marvel path is one which describes

the location of an object by specifying all if the objects hierarchically between the object in question and the current instance (see section 2). The current instance must be hierarchically above the objects being dealt with.

Marvel's ordering of objects is then used with this Marvel path to descend the object hierarchy to the correct object. In practice, the limiting factor of the line-oriented interface has been how fast a user can figure out what the current status of the system is, and what to do next.

The preserved ascii format objectbase must be read into memory whenever Marvel starts up. The initial startup time is a one time cost of $O(n^2)$ to generate the links and tables mentioned above. This would be reduced to $O(n)$ with a binary copy of the objectbase being stored. Still, it is a small cost for hours of speed. For more details of the Marvel objectbase, see [Sok89].

The time complexity design constraints mentioned in section 7.1 are realized in the graphics interface, but not the line oriented interface. In reality, however, it is unclear just how many objects must be in an objectbase in order for the line-oriented interface search time to become significant factor compared to the physical disk transfer time. This is so because objects tend to represent large chunks of data in software systems. A rigorous time analysis is difficult here, because there are so many possible factors, such as how much information is being transferred, how large the objectbase is, the speed of disk access versus in memory searches, and so forth. In practice, the limiting factor of the line oriented interface has been how fast a user can figure out what the current status of the system is, and what to do next.

7.3.3 Graphics interface

The graphics interface for the Organ command set is implemented in a consistent, event based style. All commands except add allow multiple invocations without restarting the command. Furthermore, full information on what actions (mouse picks) the system is waiting on are displayed. After each cycle of a command, the resulting objectbase is updated and redisplayed. User input for the entire command set is via the mouse, except for new names necessary for add and rename.

7.3.4 Line Oriented Interface

The line interface of the Organ commands follow a simple, single action model. Command lines are as follows:

```
add [-a <attribute-name>] <instance-name>
rename <target-object> <instance-name>
delete <target-object>
move <initial-object> <target-object>
copy <initial-object> <target-object>
join <initial-object> <target-object>
```

For <object> and <target-object>, fully qualified Marvel paths must be supplied. For example, the syntax for the join command resulting in figure 8 is:

```
join software_systems/marvel/lib/load software_systems/marvel/lib/dbman
```

7.4 Organ Completeness

The Organ command set provides most of the ability to do full scale reorganization to a large system. Below are features Organ does not have that seemed desirable during the implementation and testing of Organ.

Just as the Unix commands `mv`, `cp`, `ln` and so forth can be thought of as building blocks for shell scripts that do more complex things, the Organ commands can be imagined as building blocks for Marvel command scripts. While Marvel currently only supports command scripts upon startup, a generalized facility for executing Marvel command scripts is a simple extension. With such a facility, Organ commands could be combined to be more powerful and automatic.

Organ lacks a general undo facility. While most of the commands can effectively be undone by repetition of a reversed set of commands (for example, a `join` can be undone by a series of `moves`), there is no facility to undelete an object. The need to answer a query before a `delete` takes place is a partial solution to this.

Furthermore, Organ lacks capability to reorganize parts of an object, such as it's attributes, without reorganizing the whole object. Use for such a facility is certainly conceivable.

8 Design: User Interfaces

This section discusses the design of the user interfaces for Marvel, Organ and the Marvelizer. Emphasis is placed on the graphical interface. This discussion is included because the graphical interface has a large impact on the usability and speed of Organ, rather than as the implementation of some research on user interfaces.

8.1 Design Requirements of the User Interface

In order to satisfy a variety of hardware facilities, the user interface for Organ is made up of two separate, equivalent parts. The first is a graphics interface, and the second is a line oriented text interface. The command sets are almost identical, with changes only where one environment or the other makes certain commands inappropriate

8.2 Graphics Interface

The Marvel graphics interface is a menu and mouse driven interface built with X–11 version 3 Xlib primitives. No specific graphics toolkits are utilized. The main purpose of the interface is to facilitate a visualization of the Marvel objectbase, and a platform for the efficient implementation of the Organ command set.

An attempt has been made to stay clear of the concerns of the "leading edge" in user interface research, but rather focus on a simplistic, but fully functional interface. It is the author's opinion that much work on user interfaces is highly personal, rather then based on some set of quantitative measures, and thus this issue has been left alone as much as possible. "Wizzy" features have been added only where they have a direct influence on the usability of Marvel and Organ.

The interface is designed to be static, and is not especially tolerant of heavy user manipulation, such as resizing, iconization, and lowering of windows. Since all information displayed is disposable, a redraw can always return a user to a state where (s)he was previous to some window manager operations. Most importantly, such window operations can not cause a loss of data.

The following sections describe the layout of the various windows, and then the

`print` and `browse` commands, which both heavily utilize the graphics interface.

8.2.1 Window Layout

The graphics interface contains five windows, as indicated in figure 9.

There is a window called *menu* containing all commands built into Marvel (including the Organ commands and the Marvelizer) in a static menu. Submenus overlay this menu for appropriate commands. There is another window for Marvel rules called *rules*. The entries in this menu change dynamically as the user loads, unloads and merges strategies (see [BS88]). The largest window is the *display* window, which shows the entire objectbase upon startup. There is a browser for the *display* window, and objects can be directly picked when appropriate. Below the *display* window is the *text* window, and above it is a long thin *status* window. These windows are all statically placed, and of fixed[7] size.

1. Status Window

   The status window (Figure 10) describes the system's current status. If a command is being executed, its name appears in the center of the status window. The version of Marvel being used appears in the right part of the window, and the current object (section 7.2.1) appears in the left part of the window.

2. Display Window

   The display window provides a graphical drawing board for display of the objectbase[8]. The display is created in a two pass preorder traverse of the

[7]The configuration is trivial to change in the code, of course.

[8]In the future it will also be used for display of rule chaining.

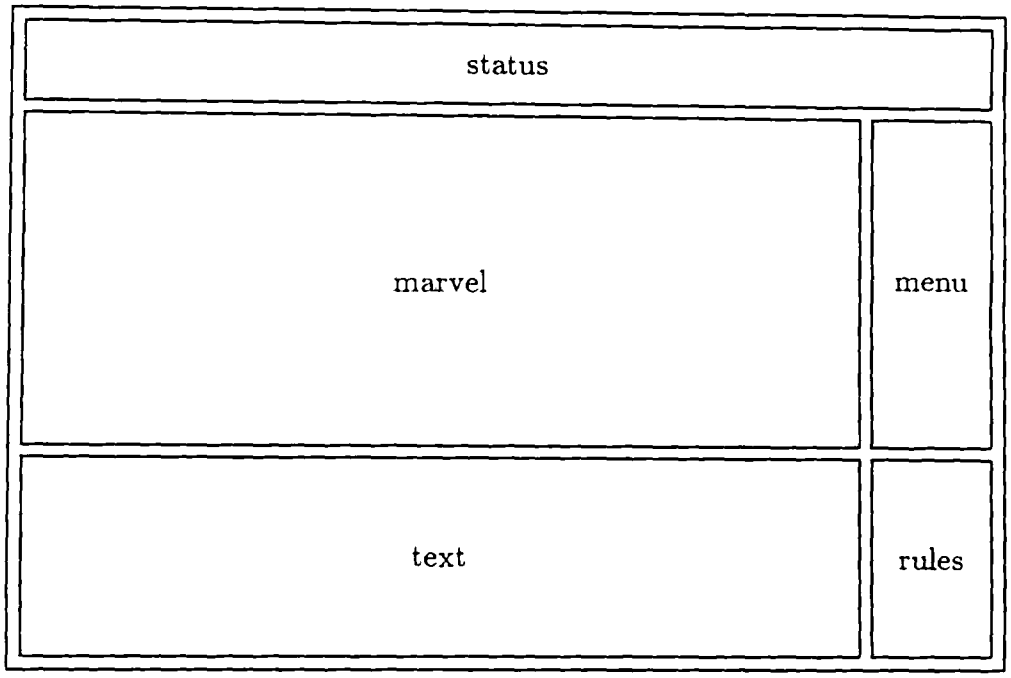

Figure 9: Layout of the Marvel Graphics Interface

current version	current command	current object

Figure 10: The Status Window

objectbase. No disk access is required, as all crucial class and attribute information is always in memory. Information displayed is strictly hierarchical, and the internal computations are done via levels, starting at zero for the root of the current display. Since zooming in and out and panning are allowed, the number of levels in the display and the display's root change dynamically.

There is a concept of a *valid* display. The display must be valid for the browser and Organ commands to operate. In general, the display is always valid, because all commands which update the objectbase cause a redisplay. If an Organ command fails with some sort of internal error, it is possible the display will be left invalid. To re-validate it, a `print all` command must be executed.

When a display must be redrawn, the procedure is as follows. Two preorder traversals of the objectbase to be displayed are performed. The first pass generates all the information needed for calculation of physically where to display each instance. This information is kept on a display structure that is directly accessible via level information available during the traversal, and in each instance. The size of this structure is just a fraction of the size of the entire objectbase, so no heavy tolls in space are extracted. The second pass actually draws the information on the screen, using the precalculated information in the display structure to decide what to draw where. Using this methodology, complex objectbases can be rapidly displayed. The time complexity of display is $O(2n)$, where n is the number of objects in the tree being displayed. This root object is should be a root for a small subset of the objectbase, as display of the entire objectbase often overly crowds the display.

Access to all components of the display is achieved with *no* searching. This

speed minimizes the waiting time for display updates, and browsing. This is done by first transforming the y coordinate of a display mouse pick into a level (via a simple linear transform) and then using this level to index into the appropriate level of the display structure. The actual instance is then found by a simple linear transformation of the x coordinate of that same mouse pick, to get the position of the instance in question in the calculated level. This position is then used as a second index into the display structure, that contains a pointer to the actual instance in the objectbase.

Figure 11 depicts an objectbase in the display window. The structure of the objectbase shown matches that for the *C/Marvel* project shown in figure 1. This example objectbase contains two complete versions of Marvel, or roughly 525 objects. When objects get too close together, as in the last two levels of figure 11, object names are left out.

3. Text Window

The text window is used to communicate with the user. Both input and output pass through this window. Output is paged for operations which have more then one screens' worth of output, such as many print commands, and the help commands. A menu pick allows repeat viewing of all next and previous pages. For interactive commands such as the Marvelizer, output is paged to keep important information from running off the screen before the user can read it.

A complete scrolling facility is currently under development as part of a larger project by a project student, and should be in place by the close of this semester.

All keyboard input for the graphics user interface comes from the text window.

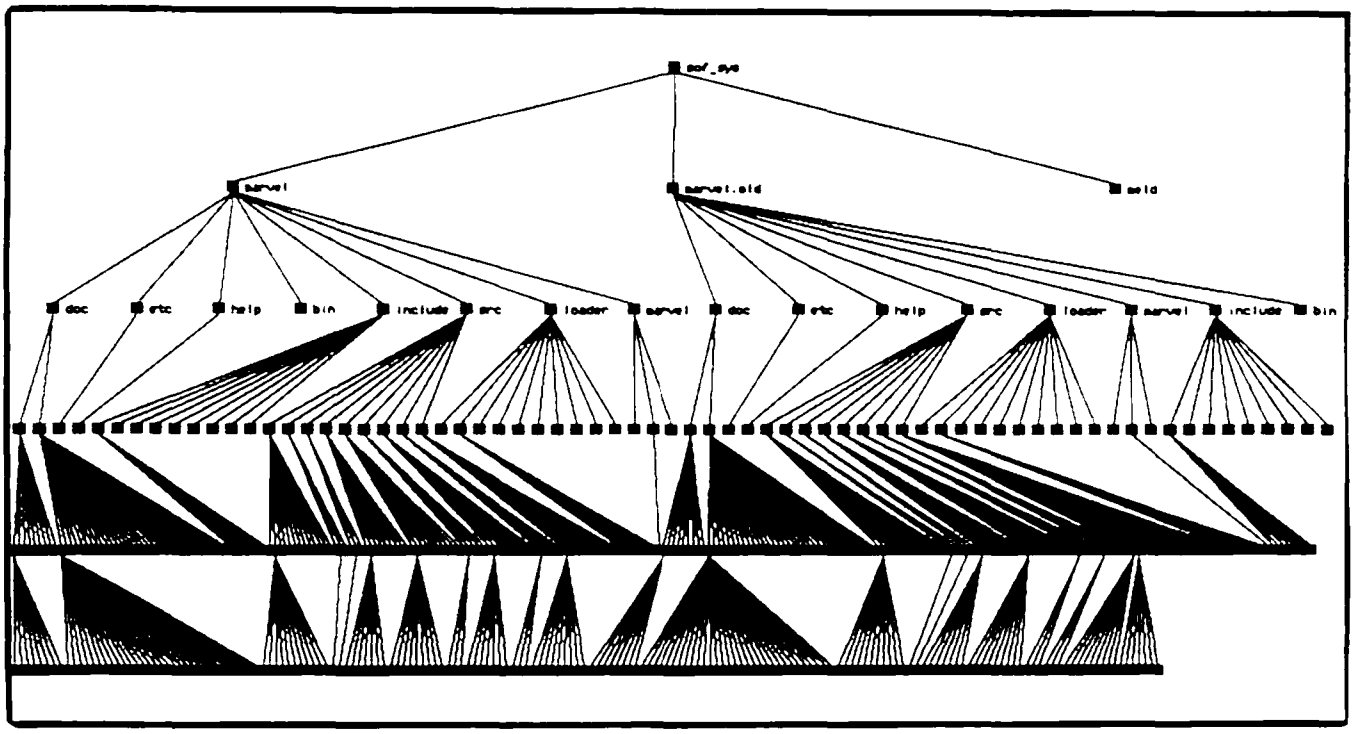

Figure 11: The Display Window

A cursor will appear when the system is waiting for input, at which point all keyboard events are focused into that window. Thus, the mouse need not be moved. A carriage return, Ctrl X, or Ctrl C ends an input string, the later two resulting in empty strings.

4. Menu Window

   Menus in the graphics interface are all static. This is to allow the user to focus on the real problem at hand, namely how to develop some software, rather then how to get through a complex menu system. All the functions available at any given moment are always visible. Figure 12 depicts the main menu. Submenus overlay the main menu at appropriate times. Submenus are described in sections 8.2.2 and 8.2.3.

5. Rules Window

   The currently available rules are accessible in the rules window. This menu is dynamic, in that it changes whenever the current set of rules changes due to a `load`, `unload` or `merge` command; however it is static in the sense that it is always available without a mouse pick to bring it up (or put it down). Figure 13 shows the rule window when the *AllTools* strategy of *C/Marvel* is loaded.

8.2.2 Print Command

The `print` command facilitates printing of the objectbase, and the currently available rules, relations, and class structures. Both the text window and the display window are used. See figure 14 for all the options available with `print`.

In order to display the entire objectbase, choose the `all` option. The objectbase

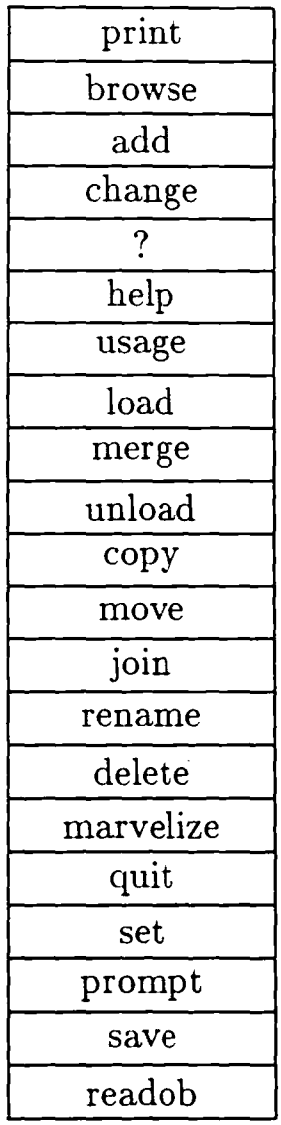

Figure 12: The Main Marvel Command Menu.

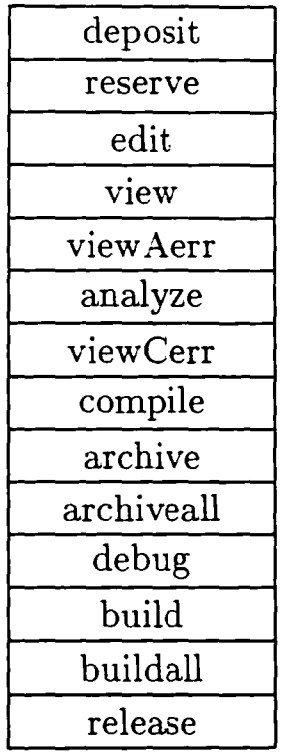

Figure 13: The C/Marvel Rules Menu.

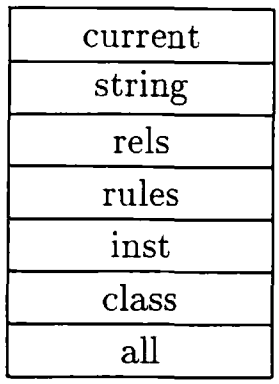

Figure 14: Print Command Options.

is displayed, and the text window is put into page mode with a textual version of the displayed objectbase. The functionality of the other options maps directly to the single letter options provided in the line oriented user interface, and is described in [BS88].

8.2.3 The Browser

The browser is a facility only available in the graphics interface. The browser allows a user to visually browse the objectbase, getting information about objects in the display as desired. In order to enter the browser, the display must be valid. In general all objects mentioned in the descriptions below get picked with the mouse in the display window. The logical boundaries of an object in the display window include the imaginary boundary half way between the object and it's neighboring objects. A mouse pick anywhere in this region is acceptable to select the object. Following is a description of the specific facilities available in the browser.

Zoomin `Zoomin` shows details of a particular part of the display. The picked instance will become the new root of the display. The display is recalculated

and redisplayed.

Zoomout `Zoomout` enlarges the current display by one level of hierarchy. This is accomplished by finding the parent object of the picked object, and making it become the new root of the display. The display is then recalculated and redisplayed.

Pan `Pan` pans left or right from a picked instance in the objectbase. This is accomplished by finding the object picked in the display structure, then making the previous (left) or next (right) element in the display structure the new root. The display is then recalculated and redisplayed. A limitation of pan is that panning can take place only between objects currently in the display.

Info `Info` gets information about a particular object in the display. Including the name of the object, all it's attributes are listed, as well as it's owner attribute and master class.

Done `Done` exits the browser.

Figure 15 shows part of a *C/Marvel* objectbase being browsed. Information about the displayed root is given in the text window. Figure 16 shows the same objectbase, after zooming in on object *organ*. Information on object *organ* is given in the text window.

8.2.4 Change Command

The line oriented syntax of the change command is awkward, because fully qualified Marvel paths 7.3.2 must be specified to describe a unique object. In the graphics interface, this is reduced to a simple pick of an instance in the display.

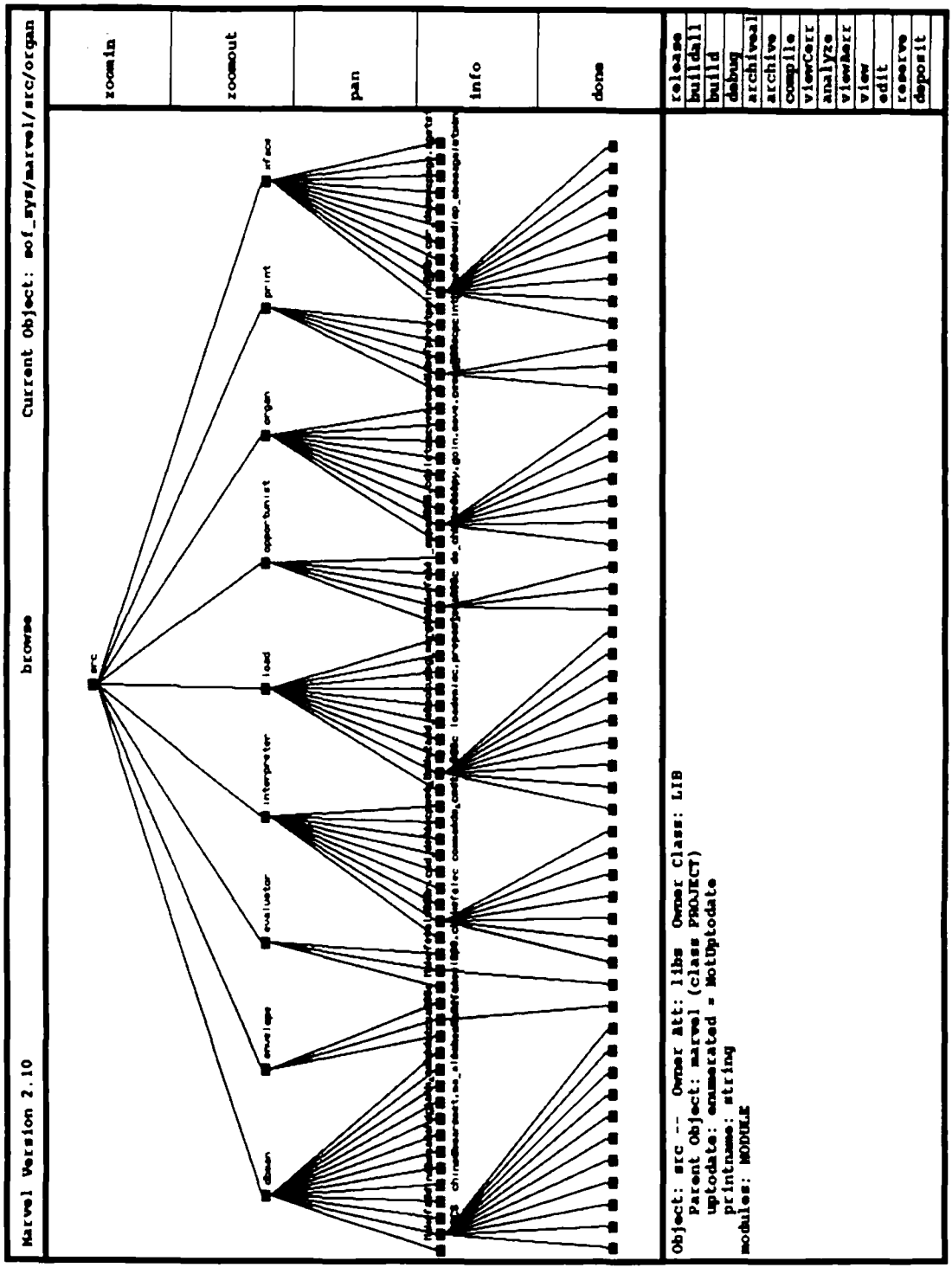

Figure 15: The browser, showing *info*.

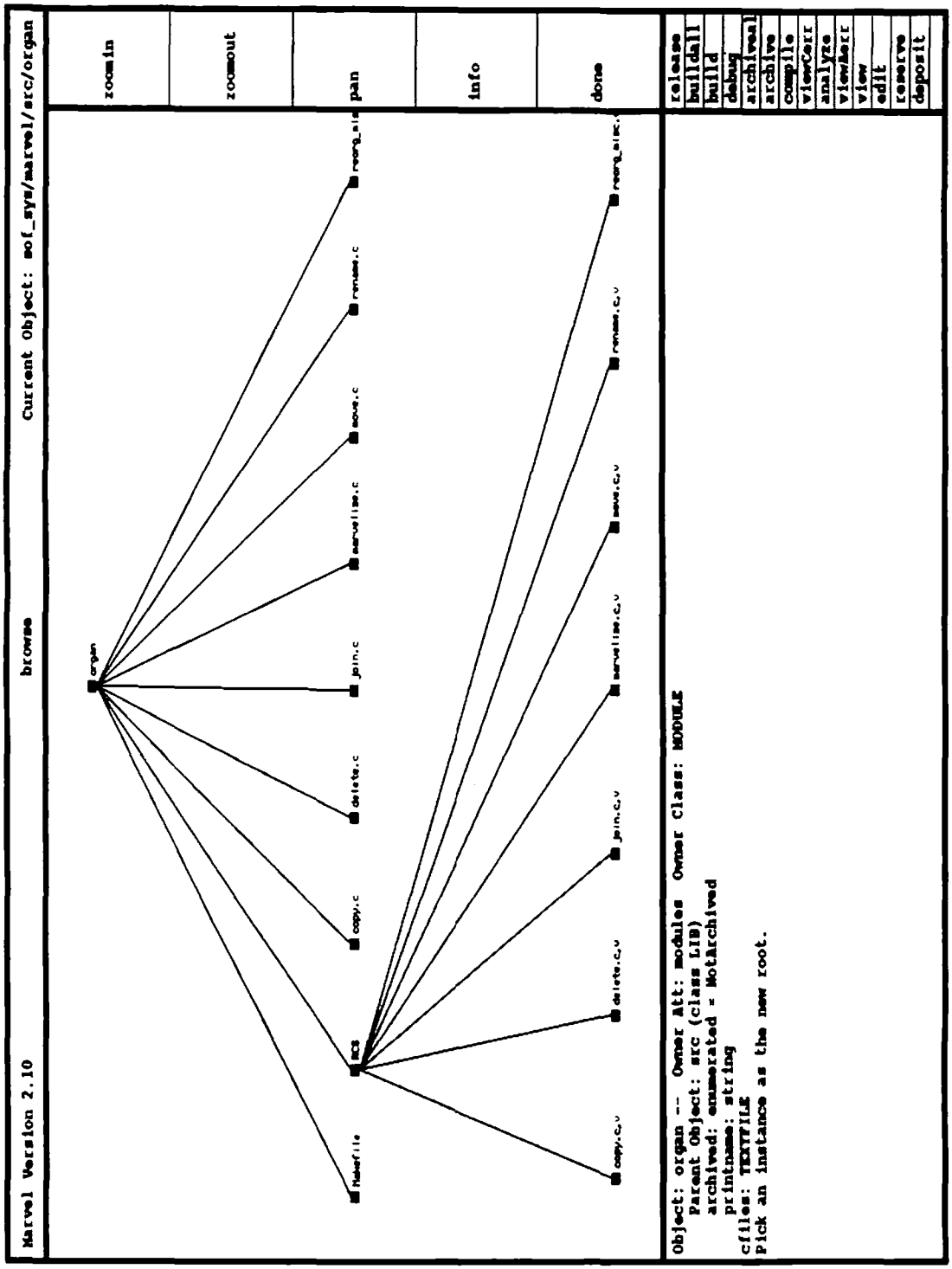

Figure 16: The browser, showing *info* and *zoomin*.

8.2.5 Commands Which are not Relevant in the Graphics Interface

Some built in Marvel commands do not have much meaning in the graphics interface. These include `set`, `?` and `Usage`. `Set` sets various settings for the line oriented print facility, thus is not often used in the graphics interface. However, these settings are reflected in the `text` window. `?` prints a list of commands available at the current time. Since the static menus are always visible, `?` is meaningless. `Usage` is of limited use, as the graphics interface of all the commands are interactive to the point where either they just need to be chosen, or if extra input is necessary, full explanation is provided on a step by step basis. Usage always prints out usage of a command in line oriented mode, if the command is different in the two interfaces. The usage of a command in the graphics mode can always be found by just execting the command. The system will give instructions. Usage is not intended to provide explaination of a command (that is the purpose of `help`, so this is acceptable.

8.3 Line Oriented Interface

The line oriented interface contains all the above commands, except as noted, with Unix like command line options rather then graphical interactions. A major difficulty in the line oriented interface comes about in uniquely identifying the objects in question. This is difficult because names are not necessarily unique within the objectbase. Thus, fully qualified Marvel paths 7.3.2 must be supplied for each object. Since Marvel is designed to map to the Unix file system, names are expected in path form from a uniquely identifying root. The line oriented interface is further documented in [BS88].

This search methodology for finding objects is breadth first. The worst case

search time will be on the order of the size of the objectbase. In general, the search speed in this case is much faster, for two reasons. First, most commands do not need to search near the leaves of the tree, which tend to be much more populated then the internal nodes, and therefore the true time bottleneck. Second, the instances of each class in the objectbase are maintained in alphabetical order, to avoid unnecessary search time. Section 7.3.2 discusses this issue in more detail.

As a general note on speed, many of the operations defined above have the potential to copy large (physical) quantities of data. Because of this, analyses of the sort used to develop the worst case figures above break down, and the linear factor of code movement and operating system overhead becomes a greatly overriding term in the net time complexity of each operation.

It is furthermore worth noting that in addition to raw search speed, the graphics interface has a more important factor which makes it *seem* faster then the line interface. This is the look and feel of the environment, which accounts for many graphics interfaces "feeling" faster then their equally functional line interfaces.

9 Future Research Areas

In the course of this research, many areas for future research have been discovered. The subjects include weaknesses in Marvel which have been made clear because of the addition of Organ and the Marvelizer, and weaknesses in Organ and the Marvelizer themselves. Research into graphics interfaces is certainly another area that has been ignored in this thesis. Use of Marvel has shown the need to research the interface question in Marvel in more detail, however the ad hoc interface created for Marvel is unquestionably an enhancement.

9.1 Marvel

Marvel does not fully utilize information provided by medium attributes. These are attributes that describe things on the granularity of a file. A set_of *medium* attribute might be very helpful to ease the awkwardness of describing the structure of many software projects. Such an attribute would alleviate the proliferation of directories needed to physically define a structure such as a module, that tends to contain a set of files or procedures that need not all be separated by directories, but can rather just be files. In the Marvel objectbase, communication with the file system is strictly via large attribute (directory) names, rather then large and medium attribute names.

Furthermore, the MSL language is a powerful language for defining environments, but is quite awkward to use. A better interface to MSL would ease future development of dynamic immigration and reorganization, as it would provide a basis for class structure modification.

There are many, many other enhancements that would make Marvel better, but most of them are not as directly related to this thesis as those above, and thus are not mentioned.

9.2 Organ

Organ is a *static* reorganizer. Static refers to reorganization that takes place within a particular class structure. The problem is when reorganization of the class structure is desired in addition to reorganization of the actual objectbase. This is called *dynamic* reorganization, or schema evolution. While static reorganization is unquestionably an important facility which has not been realized in other SDEs,

full dynamic capabilities are needed to provide all the facilities of the underlying operating system, and more traditional relational database facilities.

Organ focuses on programming in the large, versus programming in the many. Thus, as with Marvel, the entire issue of multiple users and concurrency has been left as an open research issue.

Organ (and the Marvelizer) were designed with the thought of moving to a "real" objectbase and objectbase management system. The main features that they depend upon are the class and attribute structure enforced by the current objectbase. With an objectbase system with similar facilities, the new tools should be easily ported.

9.3 The Marvelizer

While the Marvelizer is a powerful immigration tool, it provides no facilities for schema evolution. If a project is being developed with an awkward physical layout on disk, the Marvelizer will not greatly improve matters. Solutions to schema evolution in object–oriented databases have been presented by Skarra and Zdonik [SZ86]. Whether these solutions are applicable to schema evolution in Marvel remain yet to be seen.

9.4 Schema Evolution

Some possible enhancements of the Marvelizer and Organ that would provide facilities for schema evolution tool are presented now.

- A class and attribute editor for Marvel's objectbase structure can be created. Such an editor could utilize the user interface "drawing board" mentioned earlier and allow users to dynamically specify changes to the current class structure. In addition to graphically displaying changes, the editor would propagate the changes around the objectbase. Alternatively, a non graphical version of the editor could accept files similar to current strategy descriptions (except providing mapping from old to new classes), and merge new classes and changed classes in with existing classes.

- A *Meta-Marvelizer* for the Marvelizer can be created. Such a tool would apply a process similar to what the current Marvelizer does to the structure of a Marvel objectbase, and then perform an appropriate Marvelization on the current objectbase to convert it into the new format.

- Hooks in the Marvelizer to allow interaction with locally available tools, such a parsers, seem necessary. Such tools allow a software project to be managed at a finer granularity. A general interface to such tools is also necessary. Much more research needs to be completed on the general subject of schema evolution in object–oriented environments, and on the specific areas the above ideas mention.

- If the Marvelizer were able to generate an ad hoc, on the fly class structure, it would be able to complete successful Marvelizations in every case. Then class structures could be modified with one of the above dynamic schemes, and the user would have a perfectly customized objectbase. Of course, strategies and envelopes would have to be modified to work with this new objectbase, however a Marvel envelope language more sophisticated then the Unix shell could take care of much of this automatically. Much more research needs to

be done on this idea.

9.5 User Interfaces

The subject of user interface has been approached in a seemingly incomplete fashion here. This is because the intent of this research is not to focus on state of the art research on user interfaces (or state of the emotions, as the case may be), but rather on software organization. At this point, it is clear that there are more then emotional issues brought up by the implementation of the Marvel graphics interface.

9.6 Real Objectbase Managers

Since the class and attribute structure of Marvel is relatively standard, that part of a port to some "real" objectbase manager should not present any difficulty. However, many objectbase managers do not use the file system in a normal fashion to maintain data. This raises an issue of usability of COTS tools, that can not be determined generally before examination of specific objectbases. If we adopt an objectbase manager which is "real" only in the sense that it provides concurrency control, and an external object model based upon the file system, rather then a sophisticated object manager, it is unclear that we will have an improvement over what we currently have.

10 Conclusions

Marvel, the Marvelizer and Organ together provide a powerful integrated software development environment which is easily tailored to the development of a large variety of software systems. The environment is designed to grow as the project being developed grows, through environment changes and Organ's reorganization capabilities.

The Marvelizer is an immigration tool for software systems. This tool allows arbitrarily complex systems to be immigrated into Marvel after the creation of an appropriate Marvel environment. The claim of the Marvelizer is that it is much simpler to create a Marvel environment and Marvelize a system, then do the migration process by hand. This was demonstrated by the creation of the C/Marvel environment, and the actual Marvelization of Marvel.

If a project was being developed with an awkward physical layout on the disk, the Marvelizer will not greatly improve matters. As users desire to deviate the objectbase structure from the original file system layout, the need for a schema evolution tool appears. I envision such a tool as a *Meta-Marvelizer*, a Marvelizer for Marvel's objectbase structures. Schema evolution here differs from dynamic reorganization mainly in the timing of its application.

Organ is a facility that provides core software reorganization tools in an integrated software development environment (Marvel). The core commands allow object addition, copying, moving, joining, deleting and renaming. This functionality is provided in a sophisticated, easy to use fashion, in contrast to previous ad hoc methods. Sophistication comes because of a graphical interface which gives the user a *picture* of the software being manipulated, and then allows the user to work

by directly manipulating that picture. Organ is a *static* reorganizer, and does not support *dynamic* reorganization, which involves reorganization of the objectbase's class structure, or schema evolution. Static reorganization is unquestionably an important facility which has not been realized in other SDEs. However, dynamic reorganization will become necessary in the future to support a complete data migration facility.

Acknowledgments

Work on the initial implementations of Marvel are detailed in [BK88]. Marvel 2.01, which represents an significant re-implementation of the initial work, was completed by Naser S. Barghouti and myself in November 1988. Organ, the Marvelizer, and the supporting graphics interface were completed in February 1989 by myself. The current release of Marvel is now Marvel 2.10.

References

[AH87] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA '87 Proceedings*, pages 430–440. ACM, October 1987. Special issue of SIGPlan Notices, 22(12), December 1987.

[BK88] Naser S. Barghouti and Gail E. Kaiser. Implementation of a knowledge-based programming environment. In *21st Annual Hawaii International Conference on System Sciences*, volume II, pages 54–63, Kona HI, January 1988.

[BL86] Arthur J. Benjamin and Karl M. Lew. A visual tool for managing relational databases. In *1986 International Conference on Data Engineering*, pages 661–668. IEEE Computer Society, February 1986.

[BS88] Naser S. Barghouti and Michael H. Sokolsky. *Marvel Users Manual, Version 2.01, CUCS-371-88.* Columbia University Department of Computer Science, November 1988.

[Fel79] S.I. Feldman. Make – a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, April 1979.

[Hew86] Hewlett Packard Laboratories, Palo Alto CA. *SPMS Software Project Management System User's Manual*, version 1.0 edition, 1986.

[KF87] Gail E. Kaiser and Peter H. Feiler. Intelligent assistance without artificial intelligence. In *32nd IEEE Computer Society International Conference*, pages 236–241, San Francisco CA, February 1987. IEEE Computer Society Press.

[KFP88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, pages 40–49, May 1988.

[KP87] Gail E. Kaiser and Dewayne E. Perry. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Conference on Software Maintenance*, pages 108–114, Austin TX, September 1987.

[LRPC84] David B. Leblang and Jr. Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. In *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh, April 1984. Special issue of SIGPlan Notices, 19(5), May 1984.

[ML88] Axel Mahler and Andreas Lampen. An integrated toolset for engineering software configurations. In *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 191–199, Boston, MA, November 1988. ACM. Special issue of SIGPlan Notices, 24(2), February 1989.

[OHK87] Patrick O'Brien, Daniel C. Halbert, and Michael F. Kilian. The trellis programming environment. In *OOPSLA '87 Proceedings*, pages 91–102. ACM, October 1987. Special issue of SIGPlan Notices, 22(12), December 1987.

[PK87] Dewayne E. Perry and Gail E. Kaiser. Infuse: A tool for automatically managing and coordinating source changes in large systems. In *ACM 15th Annual Computer Science Conference*, pages 292–299, St. Louis MO, February 1987.

[PS87] Jason D. Penney and Jacob Stein. Class modification in the gemstone object-oriented dbms. In *OOPSLA '87 Proceedings*, pages 111–117. ACM, October 1987. Special issue of SIGPlan Notices, 22(12), December 1987.

[Rei84] Steven P. Reiss. Graphical program development with pecan program development systems. In *ACM SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, Pittsburgh PA, April 1984. ACM. Special issue of SIGPlan Notices, 19(5), May 1984.

[RW89] Larry Rowe and Sharon Wensel, editors. *1989 ACM SIGMOD Workshop on Software CAD Databases*, Napa CA, February 1989.

[SKHA86] Barbara J. Staudt, Charles W. Krueger, A.N. Habermann, and Vincenzo Ambriola. The gandalf system reference manuals. Technical Report CMU-CS-86-130, Carnegie Mellon University, Department of Computer Science, May 1986.

[Sok89] Michael H. Sokolsky. Marvel implementation guide. Technical Report CUCS-428-89, Columbia University in the City of New York, 1989.

[Sun86] Sun Microsystems. *Unix Reference Manuals*, sun os version 3.5 edition. July 1986.

[Sun88] Sun Microsystems, Inc, Mountain View CA. *Network Software Environment: Reference Manual*, version 1.1 edition, October 1988.

[SZ86] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA '86 Proceedings*, pages 483–494. ACM, October 1986. Special issue of SIGPlan Notices, 21(11), November 1986.

A *C/Marvel Objectbase*

This appendix contains objectbase definitions for *C/Marvel*.

```
STRATEGY: objectbase_def;
Imports: none;
Exports: all;

ObjectBase:
GROUP :: superclass: ENTITY:
printname : string ;
status : (Active,NotActive) = "Active";
projects : set_of PROJECT ;
END


PROJECT :: superclass: ENTITY:
printname : string ;
status : (Release,Maintenance,Development) = "Development";
allbuilt : (Allbuilt,NotAllbuilt) = "NotAllbuilt";
programs : set_of PROGRAM ;
libs : set_of LIB ;
docs : set_of DOC ;
bins : set_of BIN ;
incs : set_of INC ;
END
```

```
PROGRAM :: superclass: ENTITY:
printname : string ;
built : (Built,NotBuilt) = "NotBuilt";
cfiles : set_of TEXTFILE ;
object : binary ;
END


LIB :: superclass: ENTITY:
printname : string ;
uptodate : (Uptodate,NotUptodate) = "NotUptodate";
modules : set_of MODULE ;
libl : binary ;
END


MODULE :: superclass: ENTITY:
printname : string ;
archived : (Archived,NotArchived) = "NotArchived";
cfiles : set_of TEXTFILE ;
END


TEXTFILE :: superclass: ENTITY:
```

```
printname : string ;
compiled : (Compiled,NotCompiled) = "NotCompiled";
analyzed : (Analyzed,NotAnalyzed) = "NotAnalyzed";
reserved : (Reserved,NotReserved) = "NotReserved";
versions : set_of VERSION ;
file : text ;
END


VERSION :: superclass: ENTITY:
printname : string ;
vfile : text ;
END


DOC :: superclass: ENTITY:
printname : string ;
documents : set_of DOCUMENT ;
END


DOCUMENT :: superclass: ENTITY:
printname : string ;
formatted : (Formatted,NotFormatted) = "NotFormatted";
files : set_of TEXTFILE ;
```

```
format : binary ;
END


INC :: superclass: ENTITY:
printname : string ;
hfiles : set_of TEXTFILE ;
END


BIN :: superclass: ENTITY:
printname : string ;
executable : binary ;
END
END ObjectBase
```

B *C/Marvel Rules: AllTools*

This appendix contains a strategy for *C/Marvel* called AllTools.

```
STRATEGY: all_tools;
Imports: objectbase_def;
Exports: all;

ObjectBase:
RELEASE :: superclass: TOOL:
release : string = "release";
END
BUILDALL :: superclass: TOOL:
buildall : string = "buildall";
END
BUILD :: superclass: TOOL:
build : string = "build";
END
DEBUG :: superclass: TOOL:
debug : string = "debug";
END
ARCHIVEALL :: superclass: TOOL:
archiveall : string = "archiveall";
END
ARCHIVE :: superclass: TOOL:
archive : string = "archive";
END
COMPILE :: superclass: TOOL:
```

```
compile : string = "compile";
END
VIEWCERR :: superclass: TOOL:
viewCerr : string = "viewCerr";
END
ANALYZE :: superclass: TOOL:
analyze : string = "analyze";
END
VIEWAERR :: superclass: TOOL:
viewAerr : string = "viewAerr";
END
EDIT :: superclass: TOOL:
edit : string = "editor";
END
VIEW :: superclass: TOOL:
view : string = "viewer";
END
RESERVE :: superclass: TOOL:
reserve : string = "reserve";
END
DEPOSIT :: superclass: TOOL:
deposit : string = "deposit";
END
END ObjectBase


rules
```

```
release[?t:PROJECT]:
forall PROGRAM ?p
suchthat
(member [?t.programs ?p])
:
(?p.allbuilt = Allbuilt)
{ RELEASE release ?t }
(?t.status = Release);
(?t.status = Maintenance);
(?t.status = Development);
```

```
buildall[?t:PROJECT]:
forall PROGRAM ?p
suchthat
(member [?t.programs ?p])
:
(?p.built = Built)
{ BUILDALL buildall ?t }
(?t.allbuilt = Allbuilt);
(?t.allbuilt = NotAllbuilt);
```

```
build[?p:PROGRAM]:
suchthat
:
{ BUILD build ?p }
(?p.built = Built);
(?p.built = NotBuilt);



debug[?p:PROGRAM]:
suchthat
:
{ DEBUG debug ?p }


archiveall[?l:LIB]:
forall MODULE ?m
suchthat
(member [?l.modules ?m])
:
(?m.archived = Archived)
{ ARCHIVEALL archiveall ?l }
(?l.uptodate = Uptodate);
(?l.uptodate = NotUptodate);
```

```
archive[?m:MODULE]:
forall TEXTFILE ?f
suchthat
(member [?m.cfiles ?f])
:
and((?f.analyzed = Analyzed)(?f.compiled = Compiled))
{ ARCHIVE archive ?m }
(?m.archived = Archived);
(?m.archived = NotArchived);


compile[?f:TEXTFILE]:
suchthat
:
and((?f.analyzed = Analyzed)(?f.compiled = NotCompiled))
{ COMPILE compile ?f }
(?f.compiled = Compiled);
(?f.compiled = NotCompiled);



viewCerr[?f:TEXTFILE]:
suchthat
:
and((?f.analyzed = Analyzed)(?f.compiled = Compiled))
```

```
{ VIEWCERR viewCerr ?f }
```

```
analyze[?f:TEXTFILE]:
suchthat
:
(?f.analyzed = NotAnalyzed)
{ ANALYZE analyze ?f }
(?f.analyzed = NotAnalyzed);
(?f.analyzed = Analyzed);
```

```
viewAerr[?f:TEXTFILE]:
suchthat
:
(?f.analyzed = Analyzed)
{ VIEWAERR viewAerr ?f }
```

```
view[?f:TEXTFILE]:
```

```
suchthat
:
{ VIEW view ?f }



edit[?f:TEXTFILE]:
suchthat
:
{ EDIT edit ?f }
(?f.analyzed = NotAnalyzed)(?f.compiled = NotCompiled);



reserve[?f:TEXTFILE]:
suchthat
:
{ RESERVE reserve ?f }
(?f.reserved = Reserved)
(?f.reserved = NotReserved)



deposit[?f:TEXTFILE]:
suchthat
:
```

```
{ DEPOSIT deposit ?f }

(?f.reserved = NotReserved)
```