A Framework for Immigrating Existing Software into New Software Development Environments

Michael H. Sokolsky
Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Appeared in *Software Engineering Journal*, IEE, 6(6):435-453, November 1991.

Abstract

We have investigated the problem of *immigrating* software artifacts from one software development environment (SDE) to another for the purpose of upgrading to new SDEs as technology improves, while continuing development or maintenance of existing software systems. We first taxonomize the larger problem of *data migration*, to establish the scope of immigration. We then classify SDEs in terms of the ease of immigrating software artifacts out of the data repository of the source SDE without knowledge of its internal representation. A framework is presented for constructing automatic immigration tools as utilities provided by destination SDEs. We describe a specific immigration tool, called *Marvelizer*, that we have implemented as part of the *Marvel* SDE and discuss our experience using the tool.

Keywords: Reusability, architectural issues, integration mechanisms, knowledge-based approach, object-oriented approach, object bases.

1 Introduction

There is an enormous amount of existing software that must be maintained for many years into the future. Reuse depends on being able to incorporate existing software artifacts (including designs, source code, test cases and so forth) into new software projects. New software development environments (SDEs) may not be adopted unless it is easy to *immigrate* existing software artifacts into them. This is particularly a problem given the trend towards building SDEs on top of object bases, which seems fundamental to applying many aspects of the advancing technology but which rarely fit the original filesystem and/or database structures containing existing software artifacts [RW89, Tul88].

There are also countless existing software tools, and it seems likely that new commercial off-the-shelf (COTS) tools will continue to be developed independently of specific SDEs or standards recommendations such as PCTE [GMT86]. Many organizations are reluctant to abandon old tools due to large economic and personnel training investments and/or forego new tools that seem very promising to raise productivity, just because they do not match some SDE's tool interface. Therefore, it is important to be able to integrate COTS tools into new and existing SDEs. Most such tools assume conventional text (ascii) and binary files and/or specific formats of software artifacts.

Several environments (e.g., Istar [Dow87] and Network Software Environment (NSE) [AHM89]) have addressed the problem of providing the benefits of new SDE technology while supporting familiar COTS tools. These environments typically utilize database structures of some sort to supplement the software artifacts understood by COTS tools. To take full advantage of the new technology of such environments, these structures must be set up for existing software artifacts — and most of these environments provide some support for this. The purpose of this paper is to propose a general approach to solving this problem. Although our results are not limited to SDEs that integrate COTS tools, we have found that immigration from an SDE that supports COTS tools to another such SDE is somewhat simpler than other cases, due to the corresponding restrictions placed on the two data representations.

We begin our study of immigration by placing it in the context of the general *data migration* problem, which consists of immigration, reorganization, evolution and emigration. We then classify certain important types of SDEs with respect to their data repositories in terms of the ease of *iteration* and *navigation* over these repositories, and present a framework for immigration within and among several of these classes. We explain which cases require specialized knowledge of the source database structures and thus the construction of custom

immigration tools, and which cases can be automated without this knowledge. Our design for a general immigration tool requires that the source SDE provide some facility for iterating over the full set of software artifacts contained in its data repository, and supports a higher degree of automation when the source SDE also provides some facility for navigation over relations among the artifacts. The framework assumes that the physical data representations used by the destination SDE are known by the implementors of the immigration tool, but not necessarily by its "customers" who will use it to immigrate their existing software.

We present a specific immigration tool, called MARVELIZER, based on our framework. We have implemented MARVELIZER as part of the MARVEL SDE kernel [KFP88], for constructing or augmenting MARVEL object bases to represent existing software artifacts copied from the data repository of some source SDE. We discuss our experience using MARVELIZER for a number of practical examples, and conclude by summarizing the contributions of this work. Comparisons to related work are made throughout the paper as relevant.

A Note on Terminology

Every SDE is based on a data repository of some form, perhaps just the native filesystem, in which the software artifacts reside. New SDEs often invoke the popular buzzword "object-oriented", but there is much disagreement as to what exactly an "object", an "object base", an "object management system", or an "object-oriented database" is [Ban88]. Therefore, throughout the rest of this paper, we will use the terms "data model", "data item", "database" and "data repository" without any precise definition when referring to source and destination SDEs (other than MARVEL itself, where we define precisely what we mean by "object base"). We also use the term "class" to mean a grouping of data items, without implying any object-oriented concepts often associated with classes. We intend that these terms apply equally to software artifacts stored in filesystems, relational database management systems, object bases, etc.

It is important to emphasize that we do not claim that our results apply to general database applications; we address only software development environments, where we have some a priori knowledge of the kinds of data items and relations among data items likely to be supported.

2 Data Migration Taxonomy

Data migration is concerned with the parts of the lifecycle of software artifacts when they are moved (or copied) within a data repository or between data repositories and/or converted from one data model to another. We have identified four basic processes within data migration.

Immigration is the process of moving the contents of a source data repository to a destination data repository for the purpose of upgrading an ongoing software project to a new SDE or reusing old data items in a new project. Immigration includes the generation of new attributes and relations of the destination environment not supported in the source environment, as well as conversion of the source data formats to those required by the destination data repository.

Reorganization involves the rearranging of data within a single data repository using the same data model. Reorganization is needed to curb entropy as management objectives change, designs evolve, and/or the software system becomes more complex. A reorganization tool should provide facilities as least as powerful as typical operating system utilities for reorganizing files and directories. For example, Unix provides `mv, cp, ln, tar, cpio` and others, for moving, copying and linking, and manipulating large hierarchies. MARVEL provides the `add, delete, copy, move, rename` and `join` commands. A special case is to import a portion of one data repository into another, but within the same SDE. The Smile environment [KF87b] provides the `retrieve` command to move a selected subset of one of its data repositories into another one.

Evolution is the process of updating a data repository as its data model (schema) changes over time. This involves data translation and schema integration, which are generally believed to be solved problems for relational databases [Ne89, BLN86]. Some progress has been made for object-oriented databases [SZ86, BK87], but this is often limited to keeping old versions of the data model for accessing old data, and there are still many open questions, particularly regarding the operations associated with objects. We do not address these issues here.

Emigration consists of moving data out of a data repository, which is useful for archiving data items for later reuse or consultation.[1] Emigration might also be used as a back

[1] These uses of emigration were suggested by Dewayne Perry.

end for immigration, to convert data items to some standard format. Many VLSI design tools communicate via a common format, e.g., EDIF [Com85]. There is not yet any widely-accepted standard format for software artifacts (other than ascii text), although IDL and its instantiations such as Diana seem promising [Lam87, GWJr83].

In the general case, immigration is considerably more difficult than evolution. Evolution converts the contents of a particular data repository from one data model to another, but the same underlying physical representation scheme and data management architecture is employed, the implementor of the evolution tool understands this internal structure, and the changes to the data model are likely to be incremental — that is, the source and destination data models are strongly related. In immigration, in contrast, little or nothing may be known about the internal representations used in the source data repository, its end-user interface may not provide all the necessary access and manipulation primitives, and there may be little common ground between the source and destination data models. We have identified practical classes of source SDEs where automated immigration is feasible, as elaborated in the next section, but we make no claims regarding the general case. Note that it is impossible to automatically construct information specific to a new environment that simply is not known by the original environment, and at best it is extremely difficult if this information is implicit in the original environment, e.g., requiring understanding of the internal contents of data items; we address only the importation of information that is *explicitly* available in the source data repository.

3 Immigration Framework

We begin by listing general requirements for an immigration tool. We then categorize the classes of SDEs by identifying the forms of their data repositories, focusing on the capabilities for iteration and navigation over data items. We evaluate the matrix of possible kinds of immigration from each of these forms into the others, and introduce our design for a general immigration tool. In the following discussion, remember that "data", "data item" and "data repository" refer to software artifacts such as designs, source code, test cases and so on.

3.1 Requirements

Key requirements of an immigration tool include the following:

- The tool should be mostly automatic, requiring minimal interaction between the user and the ongoing immigration process.
- The immigration process should be fast. Speed is of course relative, but it seems reasonable that immigration should not take significantly longer than copying and rebuilding the entire software system.
- The tool should construct as many destination database structures as possible, rather than leaving these to be hand-generated.
- The tool should handle both complete source data repositories and their subparts (e.g. for reuse) with equal ease.
- The immigration process should require no understanding by the end-user of the internal implementation details of the destination data repository. It should not require any knowledge on the part of either the end-user or the immigration tool implementor of the internals of the source data repository.
- It should be easy to verify the results of the immigration process, preferably visually. There are cases where only partial automation is possible, and it should be easy for the user to determine what work remains to be done manually.

3.2 Data Repository Forms

The classes of SDEs we address include both those based upon filesystems and those that employ a database of some sort, which encompasses most practical SDEs. We exclude interpretive environments and language-based environments (.e.g, [Gol84, RT89, KKM87]), as outside the scope of our framework. Immigration into many language-based environments can be handled by parsing individual files and carrying out a batch attribution process, but evolution is a more serious problem. For example, the TransformGen [GKS86] tool, part of the Gandalf system, supports conversion of attributed syntax trees when the grammar used to generate a Gandalf editor is modified; this is accomplished with a monitoring process, to record the changes made to a grammar.

In order to analyze the necessary immigration processes between SDEs, we first classify the forms of data repositories employed by SDEs. Our primary concern is with the iteration and/or navigation facilities supported, as a means for visiting all the data items represented in the data repository, and retrieving these data items, their attributes and relations. If the source SDE does not provide facilities to visit all its data items, then there is no way to copy

them to a destination SDE without custom tools that understand the internal representation used in the source data repository. These types of immigration are beyond the scope of this paper.

We consider five major forms, called form 0 through form 4. In general, the larger numbers represent more complex data repositories, and often more sophisticated environments. We subdivide some of these forms into a and b, where b generally represents a more sophisticated form than a. Letters indicate finer grain differences than numbers.

Form 0a

A flat, unorganized group of files, all within the same directory. All data items are represented as files. Any control information is encoded in the names of the files, and control and management of the files is placed upon the user. The operating system provides an iteration scheme for accessing each file in turn, say in alphabetical order or according to time of creation or last update. This form represents the most naive way a piece of software might be developed, and is still commonplace amongst DOS operating system users, where software tools are relatively less sophisticated then those in operating systems such as Unix.

Form 0b

A filesystem directory structure that implies relations among directories and the files that they contain. A data item is represented as a file or as a directory containing component items (files). This is probably the most common form for "toolkits" whose only common knowledge is naming conventions such as special-purpose directories (e.g., bin, include, lib and man for Unix) and filename extensions (e.g., .c, .h and .a for C program development). Control and management of such systems tends to fall upon users, as with form 0a. The operating system provides a simple navigation facility for preorder traversal from selected root directories, with some capability for recognizing previously visited files and directories when links are supported.

Form 1

Form 0 above, plus specially formatted files and directories maintained by tools that are treated as part of the SDE. Thus, this form includes "private" data repositories of individual tools, such as the delta files of RCS [Tic85] or the Unix `sccs` tool, whose contents are intended to be hidden from users. Again, the operating system supports simple navigation, but the SDE must provide some means for recognizing the specially formatted files and directories.

If the same COTS tools that generated these files will be employed in the destination SDE, then they can be copied wholesale, but otherwise tool-specific conversion facilities will be required. We do not address the construction of such facilities here.

Form 2

The repositories of these SDEs include form 1 above, plus a database of connective and/or state information. This database might be manipulated by a general purpose database management system (relational databases have been employed in this fashion, e.g., Domain Software Engineering Environment (DSEE) [LJ84]) or might be specific to the SDE (NSE's ".nse_data" files [Sun88]). The database typically resides in distinguished locations in a filesystem (e.g., a user's home directory), as determined by the SDE. The filesystem component of the data repository can be navigated as in the previous forms, but must include a handle on *all* data items, aiding the extraction of their attributes and relations from the special database structures. So naming conventions are still important in this form to assist an immigration tool. There cannot be any data items represented solely within the database and not reflected by the filesystem; if not all artifacts are represented in the filesystem, then we treat the SDE as form 3a rather than form 2.

Form 3a

These SDEs have more sophisticated internal databases than those of form 2. The SDE interprets the filesystem name space, generating names that might be meaningless to users. Thus, information is more difficult to recover in a primitive form than with repositories of form 2. SDEs with this type of data repository often utilize "object–oriented" principles, but employ a rooted filesystem for byte-stream data items. The byte-stream data items are typically those required by COTS tools, and it is not necessarily the case that all data items are represented in the filesystem. Examples of form 3a include the Smile and MARVEL [KBFS88] data repositories. Since the filesystem can provide little help in determining the set of data items to be moved, or relations among these items, the SDE's user interface must provide an iteration or navigation facility that reaches all data items.

Form 3b

Form 3b is the same as form 3a, except the filesystem is not employed at all and all data items are internal to the database, stored typically in some directly accessed partition on a disk. These systems tend to be "object–oriented", since the relational model does not fit

Figure 1: Several different representations of data in a data repository. Objects in circles are in special formats controlled by COTS tools. Small ovals are directories, and small squares are files.

well with SDE requirements for retrieval of data items [Ber87]. Existing object base systems intended as SDE platforms include Cactis, Mneme and Observer [HK88, MS88, HZ87]; some ongoing SDE projects such as Arcadia plan to take advantage of such systems [TSY+88].

Form 4

Form 4 SDEs do not support either iteration or navigation via their user interfaces. Thus the SDE cannot supply the set of data items that need to be immigrated, unless of course an emigration tool is provided. We do not discuss immigration *from* form 4 SDEs further, although we do consider immigration *into* such repositories from less opaque SDEs.

3.3 Immigration Between the Forms

In this section, we analyze the prospects for immigration between each of the forms above. Table 1 shows several types of immigration. In general, forms 0 through 2 increase in sophistication as their numbers increase; for forms 0, 1 and 2, immigration from some form y to some other form z where $y > z$ does not make sense in practice, and therefore is not considered here (although this might come up during emigration). These forms are labeled with **X** in table 1. However, forms 2, 3 and 4 cannot be compared a priori with respect to

sophistication, since they are all based on a database system of some sort, with arbitrarily powerful functionality.

Since names and relations of files and directories are all that comprises the data repository in form 0, immigration from one form 0 database to another is simply a reorganization of the files and/or directories. These cases are labeled **Reorg** in the table. Form 1 adds the possibility that certain files and directories are controlled by specific COTS tools. Immigration from form 0 to form 1 thus exploits the initialization commands provided by the individual COTS tools. The automatic building of initial delta files by RCS is a good example. Any initialization beyond what the tool provides must be manual (e.g., checking in several revisions into RCS when beginning to use RCS).

For immigration from one form 1 SDE to another, either the identical tools must be employed in both environments or there must be custom facilities for converting data items from one COTS tool format to another (e.g., NSE supports conversion from RCS delta files to `sccs` delta files); we do not address these kinds of tools here, but are instead concerned with the overall data repositories. Thus, all immigration from form 0 or 1 to form 1 data repositories can be handled via a combination of reorganization tools and tool-specific initialization and conversion procedures, and is labeled **Reorg+TS** in table 1.

For immigration of form 0 and form 1 data repositories to more sophisticated forms, the immigration tool needs only the ability to navigate the files, directories and relationships implemented by the native filesystem. There are no internal database structures, and thus no need to rely on special iteration or navigation facilities that might be provided by the source SDE. We call this type of immigration the *Base Case*, labeled **Base** in table 1. We elaborate on this case in section 4.1.

The NSE `bootstrap` utility is an example of a commercial immigration tool that navigates a selected subhierarchy of the filesystem to construct its internal database. We have used `bootstrap` to immigrate the MARVEL software system (as opposed to one of its object bases) from its original form 1 filesystem data repository into NSE. NSE provides little help for immigrating from form 2 or higher data repositories.

Immigration from form 2 to form 2 and higher forms is more complex. In addition to the Base case support to navigate the filesystem component of the data repository, additional facilities are required for extracting information from the auxiliary database structures. By definition, it is possible to determine the full set of data items from the filesystem, by creating one new data item per directory and per file, with a specified relation among the data items set according to the directory hierarchy. There are no additional data items represented

	0a	0b	1	2	3a	3b	4
0a	**Reorg**	**Reorg**	**Reorg+TS**	**Base**	**Base**	**Base**	**Base**
0b	**X**	**Reorg**	**Reorg+TS**	**Base**	**Base**	**Base**	**Base**
1	**X**	**X**	**Reorg+TS**	**Base**	**Base**	**Base**	**Base**
2	**X**	**X**	**X**	**Extract**	**Extract**	**Extract**	**Extract**
3a	**X**	**X**	**X**	**Navigate**	**Navigate**	**Navigate**	**Navigate**
3b	**X**	**X**	**X**	**Navigate**	**Navigate**	**Navigate**	**Navigate**

Table 1: Immigration Cases

entirely within the database, but instead only attributes of known data items and relations among known data items. The user interface of the source SDE must provide commands to extract the attributes and relations in string form, given input identifying the desired data item that can be obtained entirely from the filesystem, such as its path name. We call this the *Extraction Case*, and it is labeled **Extract** in table 1. We introduce our design for this case in section 4.2. Note that the extraction case subsumes the base case.

For immigration from form 3 to form 2 and higher forms, it is necessary that the source SDE provide its own facility of some sort for iterating over all its data items — and preferably navigating through them to determine some basic structural relation. There is no longer any help to be obtained from the native filesystem. This facility must provide textual output, since the immigration tool can not "see" the results of a graphical browser; this is an issue given the trend for new SDEs to abandon command-line user interfaces for only graphical user interfaces, making immigration from such SDEs exceedingly difficult. We call this the *Navigation* case; it subsumes the extraction and base cases. Immigration out of form 3 is labeled **Navigate** in the table. Our design for this case is presented in section 4.3.

4 Immigration Tool Design

We have sketched five immigration cases: reorganization, tool-specific initialization/conversion, base case, extraction and navigation. Reorganization was briefly discussed in section 2, and tool-specific procedures are entirely dependent upon the particular COTS tools employed. We focus on the three remaining cases for the rest of this paper. We present abstract descriptions rather than a formal language for specifying immigration processes in the following cases, because we intend this to be a framework adaptable to a variety of SDEs, with details instantiated according to a specific SDE's philosophies and data reposi-

tory structures.

The Base case handles immigrations where all data items are represented as files or directories in the source data repository, although they can be represented in any manner in the destination data repository. However, environments that support COTS tools are likely to maintain similar files and directories in the destination data repository, perhaps with opaque filenames and extra levels of directory hierarchy.

The Navigation case is concerned with the more difficult immigrations where all data items are hidden inside the source data repository and must be accessed via end-user queries, or scripts of queries. Both Base and Navigation may be combined with Extraction, to query the SDE user interface for additional attributes and relations of already identified data items. Navigation and extraction operate by mimicking a human user of the source SDE, accessing all the data items through the query facilities of its user interface.

4.1 Base Case: File System Walk

Base case immigration handles source data repositories whose only internal database structures are those manipulated by COTS tools. Our framework supports mapping of files to data items, mapping of directories to data items, and construction of additional database contents from information derived solely from filenames and hierarchical directory structures. The mappings are defined by two kinds of specifications.

File Conversion (FC) Specifications provide information about the kinds of files that can be encountered in the source data repository, and how these map to the kinds of data items represented in the destination data repository. We refer to the latter kinds as *classes* for lack of a better term, but without any intention to imply that destination data items are "objects". These specifications specify filename patterns, generally prefixes or suffixes (e.g., filename extensions) of all the different source file types that map to the given destination classes. Patterns may also match entire filenames, but do not involve the contents of files.

```
<d-class> <s-f-pattern-1> ... <s-f-pattern-n>
```

`<d-class>` refers to a class in the destination data repository, and `<s-f-pattern>` refers to a filename or pattern matching files in the source data repository.

Directory Conversion (DC) Specifications show how directories in the source data repository map to particular classes in the destination. It is also possible to map suffixes or prefixes of directory names, as with files.

Figure 2: Left: Source Form 0b Data Repository. Right: Destination Form 3a Data Repository.

```
<d-class> <s-d-pattern-1> ... <s-d-pattern-n>
```

`<s-d-pattern>` refers to a directory name or pattern matching directories in the source data repository.

Base case immigration involves one or more passes through (some subset of) the source data repository (necessarily a file hierarchy), depending on how well that structure fits the destination data model (perhaps itself a convention for a file hierarchy). Each source file might map to a single destination data item, or to a set of data items; each source directory might map to a single destination data item, to a single data item containing one or more other data items, or to a set of data items.

Consider the simple form 0b data repository illustrated in figure 2, with directories called `input, output, body` and `do-stuff`, and files with .c, .o and a.out suffixes. To immigrate it into a form 3a data repository with **PROCEDURE** and **PROGRAM** classes (among others), the following FC and DC specifications are utilized:

1. File Conversion Specifications:

```
PROCEDURE .c .o
PROGRAM a.out
```

2. Directory Conversion Specifications:

```
PROCEDURE input output body
PROGRAM do_stuff
```

The results of immigration are shown on the right side of figure 2. In the form 3a data repository, filenames are not especially human understandable, an extra level of hierarchy has been added, and special database structures appear.

4.2 Extraction: User Interface Commands

Extraction supports immigration from data repositories with an internal database. Extraction capabilities are needed to obtain more knowledge about the data items themselves, as well as interconnections between data items other then those provided by primary navigation (filesystem walk as in the previous section or navigators as in the next section). We again have several kinds of specifications, which determine when these extraction techniques should be employed. Specifications utilize "queries" or "tools", the former being commands directly supported by the user interface of the source SDE, and the latter scripts or programs enveloping queries.

Four different kinds of specifications comprise this part of our framework, two each to handle attributes of individual data items and relations between data items, respectively.

Attribute Equivalence (AE) Specifications specify how attributes of source data items map to attributes of destination data items. Mappings are determined by applying a query (that must, by definition, return a string) to the user interface of the source SDE. The resulting string is interpreted according to the type (e.g., integer, real, string, etc.) required by the specified destination attribute to obtain its value; this assumes all the source attribute types and their string formats are known, via perusal of the source SDE's user manual. These specifications are applied to all data items in the named class.

```
<d-class> <a-name>  <--  <s-query>
```

`<a-name>` refers to a destination attribute, and `<s-query>` to a query to the source SDE. Generally, users who construct these queries must have detailed knowledge of the database schema as presented through the user interface; however, this does not require understanding of the internal data representations.

Complex Attribute Conversion (CAC) Specifications specify how attributes of source data items map to attributes of destination data items, but where the mapping is determined by applying a tool to the source SDE user interface.

```
<d-class> <a-name>  <--  <s-tool>
```

`<s-tool>` refers to tools applied to the source data repository. Tools are written by the user of the immigration tool, based on information provided by the destination SDE's user manual. There is nothing we can do if some source type is completely unsupported in the destination SDE, other than flag the problems for special handling by the user.

Relation Equivalence (RE) Specifications specify how binary relations between source data items map to binary relations between destination data items. Mappings work the same way as with attribute equivalence specifications. A particular relation can apply to all the data items in some class in the destination data repository, or to one unique item.

```
<d-class> [d-item] <relation>  <--  <s-query>
```

`[d-item]` is a unique data item in the destination SDE that has a `<relation>` specified by `<s-query>`. It is optional, and would only be specified if there is a particular query that is not applicable to all the destination data items of the `<d-class>` in question. This facility also allows immigration of data items in the source SDE that might not be members of some particular class, but can be accessed via a query.

Complex Relation Conversion (CRC) Specifications specify how N-ary relations between source data items map to N-ary relations between destination data items, or alternatively, how multi-valued binary relations are mapped. As with complex attribute conversion specifications, a tool is applied to the source SDE user interface. Again, there can either be one unique data item in the destination data repository having the specified relation, or the relation can apply to each data item in the specified destination class.

```
<d-class> [d-item] <relation>  <--  <s-tool>
```

`<s-tool>` is a tool used to derive the specified `relation`.

Consider the form 3a data repository in the left part of figure 3. The source data repository has special database structures to maintain timestamps and change logs, and hypertext-like links between code files and documentation for that code. The destination data repository has attributes called **PROCEDURE.ts**, **PROCEDURE.change** and

Figure 3: Left: Source Form 3a Data Repository. Right: Destination Form 3a Data Repository.

PROCEDURE.doclink (where **doclink** is a special type of attribute that provides a link, or bidirectional binary relation between two data items). To complete the immigration, we add the following AE and RE specifications to those of the example in section 4.1:

```
for destination data item  input:
   PROCEDURE ts 'get timestamp from input'
   PROCEDURE change 'get changelog from input'
   PROCEDURE doclink 'find doc for input'
for destination data item  output:
   PROCEDURE ts 'get timestamp from output'
   ...
for destination data item  body:
   ...
```

It is assumed here that the source SDE supports queries of the forms "get <value> from <data-item>" and "find <data-item> for <data-item>". Also, this example maps source data item names to identical destination data item names for clarity, this need not be the case, as some destination SDE might get value in changing names.

4.3 Navigation Case: Data Repository Navigation

Navigation is a fundamental necessity to provide comprehensive immigration capabilities for form 3 data repositories. There are three aspects of our framework to support navigation: one or more navigators provided by the source SDE, one-to-one mappings between equivalent

kinds of data items stored by the source and destination SDEs, and complex conversion queries or tools for more complicated mappings.

Navigators are queries or tools that return the hierarchical breakdown of data items in the source data repository. The construction of the navigator depends on the native iteration and/or navigation facilities of the source SDE's user interface. This information is used as the basis for creating data items in the destination data repository according to the class equivalence and complex class conversion specifications described below. Afterwards, the extraction specifications described above are utilized to acquire the remaining parts of the data items. If the source SDE supports only iteration, then the "navigator" is a degenerate case, and makes every data item a child of the root of the destination data repository.

Class Equivalence (CE) Specifications provide information about data items in the source data repository, and how these map to data items in the destination data repository. These appear as follows, mapping source classes into destination classes. While the source data repository might not have classes, per se, it must have groupings of some sort that can be mapped to classes — the degenerate case is every data item is in a singleton grouping.

```
<d-class> <s-class-1> ... <s-class-n>
```

`<d-class>` refers to a class in the destination data repository, and `<s-class>` refers to a grouping (class) of data items in the source data repository.

Complex Class Conversion (CCC) Specifications provide information in cases where source data items do not map well to destination data items. When the mapping between source and destination items is many-to-one, one-to-many or many-to-many, an appropriate query must already exist in the source SDE's user interface or an auxiliary tool must be created by the user.

```
<d-class> <s-class> <s-query>  or
<d-class> <s-class> <s-tool>
```

This concludes our framework for immigration between SDEs. Given the lack of a universal data model for SDE data repositories, we cannot evaluate the "completeness" of this framework. However, due to the arbitrary power of "tools" in combining user interface primitives, it is clear that all the immigrations we have described are possible if the user interface

supports the necessary access. The more important question is whether this framework will minimize the use of such tools as opposed to relatively simple single-command queries. Based on our initial experience this seems to be the case, but more empirical study is needed. In the next section, we demonstrate the practicality of our immigration framework by describing an implementation of it called MARVELIZER, a tool for immigrating software artifacts into MARVEL. We then summarize our experience using MARVELIZER.

5 MARVELIZER — Implementation and Experience

We briefly describe the relevant aspects of MARVEL in section 5.1, and then present MARVELIZER in detail. Section 5.2 presents the base case capabilities, and discusses our experience using the base case alone. Section 5.3 presents the extraction and navigation facilities, and describes our experience using this more complex version of MARVELIZER. MARVELIZER is implemented as two separate utilities, because the base case MARVELIZER was implemented (and in use) first, and later we implemented what we call Complex MARVELIZER to do extraction and navigation. The two utilities are currently invoked via two separate commands from the MARVEL user interface.

5.1 MARVEL

MARVEL is a software development environment kernel that models software processes as expert system-style rules and "enacts" these processes by forward and backward chaining among the rules. The *administrator* of a particular MARVEL environment defines the rule-based process model and the object-oriented data model in a notation called the MARVEL Strategy Language (MSL). These process and data models are then instantiated in the MARVEL kernel to form a MARVEL *Environment*. An administrator is contrasted to a user, who uses a MARVEL environment as if all process and data models were built in.

The rules determine the behavior of the environment by specifying the software development process in terms of activities and the interactions among activities. Each rule consists of a name that corresponds to an end-user command, a set of parameters indicating the expected types of argument objects, a condition that must be satisfied in order to initiate the activity, an activity represented by a tool envelope (a Unix Shell script), and one or more effects indicating the possible results of completing the activity. Which effect is actually asserted on the object base is determined by the status code returned by the envelope. Rules are thus a declarative specification of the requirements imposed by the overall software

process on individual steps of the process represented by software development activities.

The software process is automated by forward and backward chaining on the rules as follows. When the user requests a command, MARVEL uses polymorphism and inheritance to select the closest matching rule that implements the command, and checks whether its condition is satisfied. If not, it applies backward chaining in an attempt to satisfy it. Backward chaining is more complicated than in most expert systems because of the multiple effects of rules, since it is not possible to determine a priori which of the effects will be asserted before carrying out the activity. It may not be possible to satisfy the condition, in which case another rule may be tried from the inheritance precedence ordering, but if none will work, the user is informed of the problem. If the condition is satisfied, the envelope is executed outside of MARVEL, and then the object base is updated to reflect the effect of the envelope. This may satisfy the conditions of other rules, so forward chaining is applied to ensure consistency in the object base by following through with all the implications of the activity in terms of the process as a whole. Forward chaining also allows a user to be "led by the hand", to go on to the next phases of the process.

The data model given by the administrator defines the structure of a MARVEL object base, an instance of a form 3a data repository, with *classes* defining the structure of objects via multiply inherited *attributes*. Attributes may be simple entities such as integers, strings or enumerated values, text (ascii) or binary files, *sets* (to implement composite objects, i.e., containment relations), or *links* (to implement arbitrary binary relations, including one-to-many) to other objects. Objects are persistent instantiations of classes.

MARVEL maintains an in-memory object base, which contains hooks to a "hidden" filesystem. The name space for this filesystem is defined by MARVEL, with the intent of storing software artifacts in a fashion that the relevant COTS tools understand. The hidden filesystem is not intended for user perusal. We clarify this structure with an example when we discuss our base case immigration experience in the next section.

Figures 4 and 5 show the complete data model for the *C/Marvel* environment, which is used as the destination data repository when we describe our experience using MARVELIZER in the following sections. In these figures, keywords are shown in **bold**, built-in object types in *italics*, and lines beginning with "#" are comments. Classes may have one or more superclasses and any number of named attributes. **ENTITY** is the root of the class hierarchy. Each attribute is an instance of a built-in object type or an enumerated set, or indicates a containment or other relation to one or more other objects. A **set_of** attribute indicates containment by one object of an aggregate of any number of other objects, while

Figure 4: *C/Marvel*'s data model.

Figure 5: *C/Marvel*'s data model, continued.

Figure 6: A representation of *C/Marvel*'s hidden filesystem. Large circles are directories representing instantiated classes, small circles are directories representing *set* attributes, and small labels are these set attributes names, (also used as intermediate directory names). Squares map to text and binary files.

link indicates a relation between objects and/or attributes. **binary** and **text** attributes represent files in the "hidden" filesystem.

The format of the corresponding hidden filesystem maintained by *C/Marvel* is shown in figure 6. The *C/Marvel* environment consists of nineteen rules, not shown (see [CSB90]), that integrate conventional Unix programming tools for C.

Several successive versions of MARVEL have been implemented. The current version, MARVEL 2.6, consists of about 45,000 lines of C code, provides both a command-line user interface and an X11 graphical user interface, and runs on SunOS 4.0.3, Ultrix 3.1 and AIX 2.2.1. Further details of MARVEL, and our experience using it, are described in previous papers [KF87a, BK88, KFP88, KBFS88, KB88, Sok89, KBS90]. This paper is concerned only with MARVEL's data model and object base as it relates to the implementation of our two immigration tools.

5.2 MARVELIZER: Base Case Immigration

MARVELIZER is an implementation of the base case of extraction. It is implemented by the `marvelize` command in MARVEL. The base case implementation consists of approximately 1200 lines of C. We highlight the practical steps a user would take to prepare to use MARVELIZER, and then describe the process itself.

Preparation Steps

1. Prior to Marvelizing (immigrating), the MARVEL administrator must create a MARVEL data model to define the structure of the destination object base. The data model could be defined with either of two goals in mind. The MARVEL class lattice could be defined to mimic all or most of the structure of the source SDE; this of course makes Marvelization relatively easy, but would be done only when a new MARVEL environment was being developed specifically to take over the role of the source SDE. The alternative is for a MARVEL administrator to develop a data model suitable for the purposes of the new MARVEL environment, independent of whether or not the environment is planned to encompass existing data items from some other SDE that will later be Marvelized. In all our examples, here and in following sections, the data model was developed prior to the MARVELIZER tool, and thus was conceived entirely independently of the formats of any potential source SDEs.

2. Write File Conversion (FC) specifications for all the files in the original system.

3. Write Directory Conversion (DC) specifications for all directories that might be automatically converted. In verbose mode, the user will be queried for any unspecified directories. All these specifications are just as described in Section 4.1. These specifications are consulted before the immigration of each data item.

4. Choose either verbose or automatic modes to proceed, in general, automatic mode is used if the user doing the Marvelization does not want to monitor the process.

MARVELIZER algorithm

Base case immigration is accomplished via two simultaneous preorder traversals, one over the source data repository's filesystem directory structure, and the other over the MARVEL object base, starting at the destination object specified by the user. In the following description, there are notions of "current" object and class. The current object is the one being examined at some particular instant in the traversal of the destination MARVEL object base; the current class is that object's class.

When a file in the source data repository traversal is encountered, MARVELIZER checks whether the file's suffix matches a specification in the table, and if that specification's class matches either the current class (the class of the current object in the MARVEL object base), or a set attribute of the current class. In the first case the file is simply copied into the appropriate place in MARVEL's hidden filesystem space, as determined by the current object. In the second case, which gets priority in case both cases are true, a child object is hierarchically added to the current object; then the (source) file is copied to a place in the hidden filesystem determined by the new child object. Files not specified by a specification are skipped. MARVELIZER generates messages specifying those files that were skipped, with the explanatory content of these messages depending upon the verbosity mode chosen.

When a directory in the source data repository is encountered, MARVELIZER first looks to see if there is a matching specification. If so, and if the specification's class is the current class of the destination object base traversal, then a corresponding new object is added to the destination object base, as described above. Otherwise, MARVELIZER determines the set of possible classes this directory could be an instantiation of, based on the attributes of the current object in the (destination) object base traversal. If there is more than one matching attribute type or class, the user is queried (possibly skipping the directory is an option). This is the only direct interaction with the user once the process has started, and can be turned off, to only generate messages for the user to look at later.

This process continues recursively, until the traversal of one or the other data repository

is complete. If the MARVEL objectbase traversal completes first, those remaining portions of the filesystem traversal must not match MARVEL's current data model, and must be separately marvelized. If MARVEL's graphics interface is being employed, the visual display of its object base is updated after Marvelization. At this point, any software artifacts that were not successfully immigrated (i.e., were skipped) can be reMarvelized individually, by running MARVELIZER again using a different user-designated object as the starting point and a subset of the source SDE's filesystem as the source root. Such failures happen when MARVELIZER cannot recognize the structure of an existing source directory hierarchy, for example, when insufficient specifications were provided.

Base Case Experience

We have applied MARVELIZER using two data models, *C/Marvel* described above, and *DocPrep* for formatting documents with the text processor `Scribe`.

Using *C/Marvel*, we immigrated MARVEL itself with the base case MARVELIZER. The immigration process was straightforward, as we have been doing development work on MARVEL directly on top of the Unix filesystem, using a variety of COTS and custom tools, rather than using a particular SDE. Hence, the MARVEL code was in a form 1 data repository. All the tools we have been using were easily integrated, as a primary motivation for the creation of *C/Marvel* was to enable us to use a MARVEL environment to continue our own development of MARVEL.

The initial dialog with MARVELIZER for the above Marvelization is shown in figure 7; user responses are in *italics*. The user first specifies the location of the original SDE and the object in a *C/Marvel* object base to start the algorithm. MARVELIZER then requests all FC and DC specifications. Figure 8 shows the final results of Marvelizing the MARVEL system. The code for MARVEL was divided into a shared library and two programs. The programs did not quite fit the data model for *C/Marvel*, so they had to be Marvelized separately from the shared library. All together, the entire process took about 20 minutes (elapsed time) on a Sun 3/60 (3 Mips) workstation that had to copy all the code over a busy Ethernet. Derived files (.o, .a and executables) were not Marvelized, so recompilation was then necessary. We could have immigrated the derived files also, but did not do so since recompilation within MARVEL automatically initializes all its status information; otherwise, this would have had to have been hand-generated, since whether or not any given file had been compiled successfully was not explicitly available in the source data repository (although we could have written a tool to have guessed this information from file update timestamps and so forth).

```
Enter the filesystem root to be Marvelized:  /example/marvel
Enter destination class for /example/marvel:  PROJECT
(v)erbose, (q)uiet or (s)ilent mode? (any other key to exit):  v
Now enter all file suffixes for each class.
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 specific directories and the classes to immigrate them to.
Format is:
CLASS_NAME <directory-1> <directory-3> ... <directory-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 7: A dialog with MARVELIZER.

Figure 8: The results of Marvelizing MARVEL.

Using the DocPrep data model, not shown, we immigrated the MARVEL User's Manual and a 200-page PhD thesis. Both test cases were form 0 data repositories, and both immigrations went smoothly. The most interesting thing about DocPrep is that it was developed by two students as a one semester class project.[2] These students had no knowledge of either the then in-progress MARVELIZER work or the format of the PhD thesis we later immigrated successfully using their data model; the PhD thesis was written in 1985, certainly without later Marvelization in mind.

5.3 Complex MARVELIZER: Extraction and Navigation

Complex MARVELIZER is an implementation of the extraction and navigation components of immigration. It is implemented by the `c_mrvlze` and `cm` commands in MARVEL. `c_mrvlze` is invoked to enter all specifications and the navigator, to create appropriate database structures for storage of this information, and to generate a script of MARVEL commands to perform the immigration. (MARVEL has a general facility for executing batch command scripts.) `cm` is an internal MARVEL command intended for use in batch execution in these generated scripts only. It is a dispatcher that either creates a new object in MARVEL or executes a tool or query on the source SDE's user interface. Complex MARVELIZER is comprised of approximately 1500 lines of C.

Complex MARVELIZER supports run-time variables to specify the source data item being Marvelized at the current time, the corresponding MARVEL object, and that object's path in the MARVEL object base. Similar variables are supported for ancestors of these source data items and MARVEL objects. A variable that stores the name of the source SDE and any initial calling arguments is also supported. These variables are available to the user in the specification part of `c_mrvlze`.

The algorithms closely match the abstract discussion of sections 4.2 and 4.3. We now highlight the practical steps a user would take to prepare to use Complex MARVELIZER, and then describe the process itself.

Preparation Steps

1. Implement one or more navigators. These will generally be tools that envelope the query facilities supplied by the source SDE user interface.

[2] Laura Johnson and Victor Kan in E6123y Programming Environments and Software Tools, Spring 1989.

2. Write Class Equivalence (CE) and Complex Class Conversion (CCC) specifications (section 4.3) for all classes of data items in the source data repository.

3. Write Attribute Equivalence (AE) and Complex Attribute Conversion (CAC) specifications (section 4.2) for all attributes of data items in the source data repository where a non-default value is desired.

4. Write Relation Equivalence (RE) and Complex Relation Conversion (CRC) specifications (section 4.2) for all relations of data items in the source data repository where non-default relations (if any) are desired. These will map to link attributes in the MARVEL object base.

Complex MARVELIZER algorithm

1. Choose the `c_mrvlze` command in MARVEL. The next several steps outline the use of this command.

2. Input the path to invoke the source SDE, and any arguments need to execute it on the source data repository. Then input the navigator and any calling arguments. For each class in MARVEL's data model, input corresponding classes (if any) in the source data repository, and any CE and CCC specifications. Then for each chosen class, input all AE, CAC, RE and CRC specifications. `c_mrvlze` does immediate variable substitution of any MARVELIZER specific variables at this point.

3. The immigration is completed as follows. c and o represent classes and individual data items in the source data repository, respectively; C and O represent classes and newly created objects in the destination MARVEL object base, respectively.

4. Apply the navigator to create a preorder listing of the data items in the source data repository. The output of the navigator is a list in the format required as input for the next step. Each entry in the list specifies a source class, a MARVEL class, a data item name and a depth number for determining hierarchy. The depth number and the MARVEL class are used as a guide to determine how to build the resultant MARVEL objectbase.

5. Read this list, and create a MARVEL script that does the following:

   for all classes c do

find C, using CE or CCC specifications
for all objects o in c do
create a new object O in C by dispatching the `cm` command
apply AE, CAC, RE and CRC specifications, by dispatching one `cm` command for each specification, from o to O
at this point, other objects related to O are created by `cm`, facilitating immigrations that are not 1–1 mappings between source data items and destination MARVEL objects
done
done

Extraction and Navigation Experience

We have used our Complex MARVELIZER on the Smile (version 6.0) system, a C program development environment developed at Carnegie Mellon University [HN86]. Smile is a good test case for Complex MARVELIZER, because it has a hidden filesystem with a non-obvious internal structure, and a special purpose database that stores much additional information about the program, such as import and export lists for modules. Smile has module, procedure, object and datatype data items (among others). Modules are basic organizational blocks including imports and exports of items; procedures are stylized C code, where procedure parameters are specified in a Pascal-like style (called GC, the purpose is to facilitate inter-module type checking); objects are global variables; and datatypes are type and preprocessor definitions. For compiling, Smile combines all procedures, objects and datatypes (and externs based upon imports and exports) together into one large file per module, internally converts GC to C, and uses a normal C compiler for compilation.

We considered two distinct approaches to Marvelizing Smile data repositories: One with all the separate procedures, objects and datatypes represented as separate destination data items, shown completely Marvelized in figure 12, and one with each module combined into a large C file, shown completely Marvelized in figure 13. The particular Smile data repository shown contains Smile's own code, roughly 25,000 lines. The motivation to use the same source SDE for two different immigrations is to demonstrate MARVELIZER's capability to make multiple distinct transformations from the same source data model into the same

destination data model, depending upon the navigator and mapping specifications provided. Additionally, the immigration could have taken on a completely different nature if some other data model had been used.

For the two approaches, we have written two navigators, one a proper subset of the other. There is one CAC specification for the first case, and a CAC and a CCC specification for the second. The navigators are simple C programs (230 lines and 180 lines, including comments) respectively, and the other specifications are simple Unix shell scripts (36, 21 and 18 lines, respectively), shown in figures 9, 10 and 11. The dialog with MARVELIZER for the first case is shown in figure 14, and for the second in figure 15. The first Marvelization took approximately 2 hours, the second 15 minutes, both on a Sun 3/60. The inefficiency of the first case is due to the overhead of starting Smile up once for each data item; this would be greatly reduced by employing tools that call Smile with requests for more then one data item at a time. In both cases, telling the system the initial specifications took a few minutes. The two `cm` scripts generated by `c_mrvlze` for the two different approaches are 3587 and 119 lines long, respectively. The first part of the longer one of these scripts is shown in figure 16.

Note that we were careful to avoid considering the internal storage format of Smile's source data repository, in terms of hidden directories and special database structures, even though we had its source code available. In our attempt at a "blind" experiment, the second author, who is familiar with this aspect of Smile, did not assist the first author, who is not, in the Marvelization. Nothing in MARVELIZER "understands" Smile in any way.

6 Conclusion

The primary research contribution of this paper is our framework for immigration of software artifacts among software development environments. The framework applies to most practical classes of SDEs. The base case, extraction case and navigation case appear to cover all instances of immigration from one SDE to another where there is no knowledge of the internal representation of the source data repository and the source SDE has not anticipated the need for an emigration tool, but does provide a facility for iterating or navigating over the full set of data items maintained in its data repository.

Construction of auxiliary tools to aid in extraction and navigation depends on the capabilities of the source SDE's user interface and the degree of similarity between the source and destination data models; in any case, custom utilities will be required for converting the individual data formats required by one COTS tool to those required by another. We

```
# get_file: get a source SDE file.

# This script is a simple example of a tool that really just provides
# the querying facility to an SDE that does not quite have enough user
# interface power to do it itself.
#
usage="$0 <s-mod> <s-class> <s-object> <where-to-put> <s-sde w args>"

if [ $# -lt 5 ]
then
    echo $usage
    exit 1
fi

tmpfile=/tmp/getfile.temp
tmpsmilefile=smile.out

# run the smile commands into a file
echo "module $1" > $tmpfile
echo "print source declaration $2 $3 $tmpsmilefile" >> $tmpfile
echo "quit" >> $tmpfile

destfile=$4

shift; shift; shift; shift;

# copy that file to the correct place.
prog=$1; shift;
args=$*
$prog $args < $tmpfile

mv $tmpsmilefile $destfile # should be the same filesystem.
rm $tmpfile
exit 0
```

Figure 9: A CAC specification for approach 1

```
# get_big_cfile: get a Smile big C file.
#
# assumption is that an earlier script made all the big C files.
# this is done with a Smile command which could be called to be sure
# but this adds inefficiency.

usage="$0 <s-db> <s-program> <s-module> <path>"
if [ $# -lt 4 ]
then
    echo $usage
    exit 1
fi
cp $1/$2/$3/C_$3.c $4   # do the copying.
```

Figure 10: A CAC specification for approach 2

assume in this paper that the implementor of an immigration tool is highly knowledgable regarding the implementation details of the destination data repository, but relaxation of this requirement seems an interesting area for future work. Our successful experience implementing and using MARVELIZER demonstrates that our results represent a promising step in this field.

Acknowledgments

The development of MARVEL thus far has been a long, cooperative effort involving too many people to list individually. We would like to single out Naser Barghouti, who has been the driving force behind much of the pragmatic design and implementation; Peter Feiler collaborated with the second author on the initial conception of MARVEL. We would also like to thank Dan Duchamp and Susan Blockstein for reviewing drafts of this paper. MARVEL 2.6, which includes all of MARVEL (and MARVELIZER) as described in this paper, is fully documented (over 350 pages) and available for licensing to educational institutions and industrial sponsors.

A condensed version of this paper has been submitted to the conference *ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, under the title Immigration into Software Development Environments.

```
# mod_to_cfile a shell script that changes the name of a module in
# the source SDE to module.c for the target SDE.

OBJECT_MOD_FILE=/tmp/obj_mod.tmp    # must be so, for marvelizer
                                    # currently hard coded, but could
                                    # be changed.  This has nothing to
                                    # do with Smile.

usage="$0 <s-class> <m-class> <s-obj> <hier>"

if [ $# -lt 4 ]
then
    echo $usage
    exit 1
fi
echo "$1 $2 $3 $3.c $4" > $OBJECT_MOD_FILE

# here is where we would put other arbitrary marvel commands
exit 0
```

Figure 11: A CCC specification for approach 2

Figure 12: Smile in MARVEL : approach 1

Figure 13: Smile in MARVEL : approach 2

```
Source SDE with arguments:  smile /u/douglass/sokolsky/SMILE
Navigation function:  navigator $SP
Enter <o_class> for  GROUP:
Enter <o_class> for  PROJECT:
Enter <o_class> for  PROGRAM:  project
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
Enter <o_class> for  PROGRAM:
Enter <o_class> for  LIB:
Enter <o_class> for  MODULE:  module
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
Enter <o_class> for  MODULE:
Enter <o_class> for  FILE:
Enter <o_class> for  CFILE:  procedure
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
Enter <o_class> for  CFILE:  object
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
Enter <o_class> for  CFILE:
Enter <o_class> for  HFILE:  datatype
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
   ...
Getting Attribute methods for source project, dest PROGRAM
   ...
Getting Attribute methods for source procedure, dest CFILE
Enter 1 for query, 2 for tool, <cr> for default (attribute name):
Enter 1 for query, 2 for tool, <cr> for default (attribute owner):
Enter 1 for query, 2 for tool, <cr> for default (attribute timestamp):
Enter 1 for query, 2 for tool, <cr> for default (attribute reservation_status):
Enter 1 for query, 2 for tool, <cr> for default (attribute version):
Enter 1 for query, 2 for tool, <cr> for default (attribute contents): 2
Enter  <tool> for contents:  get file $SP $OBJ0 $OBJ1 $PATH
...
Getting Attribute methods for source object, dest CFILE
   ...
edit the command script? [y/n]  n
Executing queries ...
```

Figure 14: A dialog with complex MARVELIZER: approach 1

```
Source SDE with arguments:  smile /u/douglass/sokolsky/SMILE
Navigation function:  simple_navig $SP

Enter <o_class> for  GROUP:
Enter <o_class> for  PROJECT:
Enter <o_class> for  PROGRAM:  project
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  cr
Enter <o_class> for  PROGRAM:
...
Enter <o_class> for  FILE:
Enter <o_class> for  CFILE:  module
Enter 1 for query, 2 for tool, <cr> for equiv (class CFILE):  2
Enter <tool> for CFILE:  mod_to_cfile
Enter <o_class> for  CFILE:
Enter <o_class> for  HFILE:
Enter <o_class> for  DOCFILE:
   ...
Getting Attribute methods for source project, dest PROGRAM
   ...
Getting Attribute methods for source module, dest CFILE
Enter 1 for query, 2 for tool, <cr> for default (attribute name):
Enter 1 for query, 2 for tool, <cr> for default (attribute owner):
Enter 1 for query, 2 for tool, <cr> for default (attribute timestamp):
Enter 1 for query, 2 for tool, <cr> for default (attribute reservation_status):
Enter 1 for query, 2 for tool, <cr> for default (attribute version):
Enter 1 for query, 2 for tool, <cr> for default (attribute contents):  2
Enter  <tool> for contents:  get_big_cfile
Enter 1 for query, 2 for tool, <cr> for default (attribute compile_status):
Enter 1 for query, 2 for tool, <cr> for default (attribute analyze_status):
Enter 1 for query, 2 for tool, <cr> for default (attribute documentation):
edit the command script? [y/n]  n
Executing queries ...
```

Figure 15: A dialog with complex MARVELIZER: approach 2

```
#!marvel script

# class PROGRAM, original object smile
cm project PROGRAM smile 0

# class MODULE, original object CMDDATA
cm module MODULE CMDDATA 1

# class HFILE, original object SRC_AVAIL
cm datatype HFILE SRC_AVAIL 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

# class HFILE, original object SRC_BUSY
cm datatype HFILE SRC_BUSY 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

# class HFILE, original object SRC_DEP
cm datatype HFILE SRC_DEP 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

# class CFILE, original object CIlist
cm object CFILE CIlist 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

# class CFILE, original object PROMPT
cm object CFILE PROMPT 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

# class CFILE, original object crosslist
cm object CFILE crosslist 2
cm __SETATT__ contents __TOOL__ get_file $SP $OBJ0 $OBJ1 $PATH

...
```

Figure 16: The begining of a script generated by `c_mrvlze`

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

References

[AHM89] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object Management in a CASE Environment. In *11th International Conference on Software Engineering*, pages 154–163, Pittsburgh PA, May 1989.

[Ban88] Francois Bancilhon. Object-Oriented Database Systems. In *7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin TX, March 1988. ACM Press.

[Ber87] Philip A. Bernstein. Database System Support for Software Engineering. In *9th International Conference on Software Engineering*, pages 166–178, Monterey, CA, March 1987.

[BK87] Jay Banerjee and Won Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD 1987 Annual Conference*, pages 311–322. ACM Press, May 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. IEEE Computer Society.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[Com85] EDIF Steering Committee. EDIF Specification Version 1.1.0. *Electronic Design Interchange Format Steering Committee*, 1985.

[CSB90] Mara W. Cohen, Michael H. Sokolsky, and Naser S. Barghouti. Marvel 2.5 User Manual. Technical Report CUCS-498-89, Columbia University Department of Computer Science, May 1990.

[Dow87] Mark Dowson. Integrated Project Support with IStar. *IEEE Software*, 4(6):6–15, November 1987.

[GKS86] David Garlan, Charles W. Krueger, and Barbara J. Staudt. A Structural Approach to the Maintenance of Structure-Oriented Environments. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 160–170, Palo Alto CA, January 1986. *SIGPLAN Notices*, 22(1), January 1987.

[GMT86] F. Gallo, R. Minot, and M. I. Thomas. The Object Management System of PCTE as a Software Engineering Database Management System. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12–15, Palo Alto CA, January 1986. *SIGPLAN Notices*, 22(1), January 1987.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading MA, 1984.

[GWJr83] G. Goos, W. A. Wulf, A. Evans Jr., and K. J. Butle r. *DIANA – An Intermediate Language for Ada*, volume 161 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1983.

[HK88] Scott E. Hudson and Roger King. The Cactis Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, June 1988.

[HN86] A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[HZ87] Mark F. Hornick and Stanley B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Automation Systems*, 5(1):70–95, January 1987.

[KB88] Gail E. Kaiser and Naser S. Barghouti. An Expert System for Software Design and Development. In *Joint Statistical Meetings*, pages 10–19, New Orleans LA, August 1988. Invited paper.

[KBFS88] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database Support for Knowledge-Based Engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[KBS90] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel. In Bruce D. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[KF87a] Gail E. Kaiser and Peter H. Feiler. An Architecture for Intelligent Assistance in Software Development. In *9th International Conference on Software Engineering*, pages 180–188, Monterey CA, March 1987. IEEE Computer Society.

[KF87b] 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.

[KFP88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[KKM87] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, Distributed Language-Based Environments. *IEEE Software*, 4(6):58–67, November 1987.

[Lam87] David Alex Lamb. IDL: Sharing Intermediate Representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.

[LJ84] David B. Leblang and Robert P. Chase Jr. Computer-aided Software Engineering in a Distributed Workstation Environment. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, Pittsburgh PA, April 1984. *SIGPLAN Notices*, 19(5), May, 1985.

[MS88] J. Eliot B. Moss and Steven Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 298–316. Springer-Verlag, September 1988.

[Ne89] Erich Neuhold and Michael Stonebraker (editors). Future Directions in DBMS Research. *SIGMOD Record*, 18(1), March 1989.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

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

[Sok89] Michael H. Sokolsky. Data Migration in an Object-Oriented Software Development Environment. Master's thesis, Columbia University Department of Computer Science, April 1989, Technical Report CUCS-424-89.

[Sun88] Sun Microsystems, Inc., Mountain View CA. *Introduction to the NSE*, March 1988.

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

[Tic85] Walter F. Tichy. RCS – A System for Version Control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

[TSY+88] Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf. Foundations for the Arcadia Environment Architecture. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, Boston MA, November 1988. *SIGPLAN Notices*, 24(2), Februrary 1989.

[Tul88] Colin Tully, editor. *4th International Software Process Workshop: Representing and Enacting the Software Process*, Moretonhampstead Devon, UK, May 1988. ACM Press. Special issue of *Software Engineering Notes*, 14(4), June 1989.