Simplifying Data Integration: The Design of the Desert Software Development Environment

Steven P. Reiss
Department of Computer Science
Box 1910
Brown University
Providence, RI 02912
spr@cs.brown.edu
(401)-863-7641, FAX (401)-863-7657

Abstract

This paper describes the design and motivations behind the Desert environment. The Desert environment has been created to demonstrate that the facilities typically associated with expensive data integration can be provided inexpensively in an open framework. It uses three integration mechanisms: control integration, simple data integration based on fragments, and a common editor. It offers a variety of capabilities including hyperlinks and the ability to create virtual files containing only the portions of the software that are relevant to the task on hand. It does this in an open environment that is compatible with existing tools and programs. The environment currently consists of a set of support facilities including a context database, a fragment database, scanners, and a ToolTalk interface, as well as a preliminary set of programming tools including a context manager and extensions to FrameMaker to support program editing and insets for non-textual software artifacts.

1.0 Motivation

A software development environment is an integrated set of tools designed to support all phases of software engineering. Our research, first with the FIELD system and currently with Desert, attempts to explore different integrating frameworks for these tools. In the Desert environment we are attempting to augment control integration based on broadcast messaging with a simplified form of data integration and a common framework for editing. The result of these additions is an environment that can support new approaches to software development while remaining compatible with existing tools and working on existing programs.

Message-based control integration has been widely successful in programming environments, forming the backbone of such commercial environments as Sun's SPARCworks, HP's SoftBench, Digital's FUSE, and SGI's CodeVision. It allows independent programming tools to effectively communicate with each other, providing the user with the impression of a seamless environment. However, because it only integrates tools that are running, has no notion of history or time, and has limited information about the system being worked on, it cannot provide several desirable facilities and cannot easily extend to handle other aspects of software engineering [11].

The alternative to message-based integration has been data integration. Here the different tools in the environment share data through a central database. The benefits of data integration include the ability to share information between tools, the ability to identify relevant portions of a system based on semantic content, and the ability to define and maintain links between different software artifacts. Data integration, however, has not been particularly successful because of the cost of maintaining the complex database, the inability to use existing tools, and the difficulty in using such environments on existing, multi-language systems.

Our objective in developing the Desert environment has been to achieve many of the benefits of data integration without the costs. We focused our efforts on three particular features that we wanted to provide: hypertext access to software, viewing a system as a dynamic "electronic" document, and supporting existing tools and systems.

Our first objective was to provide hypertext links that interconnect all aspects of software engineering. We wanted to provide explicit links, allowing, for example, the user to select a requirement and find where in the system that requirement is met. More importantly, we wanted to support a wide variety of implicit links based on the underlying semantics of the system and on information stored by different tools. We wanted the user to be able to select a function or variable name and quickly go to its definition, its set of uses, its documentation, or the OMT [22] diagram where the object is defined. We wanted a performance analysis tool to use dynamic links to show the user where time is spent. Similarly, we wanted to link an entry in a error database to the various changes that were made to fix that error using information from the version control database.

Our second objective was to enable the programmer to view a software system as a single, dynamic document. Currently programming is done in terms of files. To add an additional parameter to a function, for example, the user must bring up editors on the function definition in a header file, the function body in a source file, and on each reference to that function in other source files. A dynamic view would allow the user to construct a virtual file containing the header declaration for the function, the function body, and the various call sites. The user could then edit this one file to effect the same change. Because the system would find all the call sites based on the appropriate semantic information, the user would be assured that all relevant calls were included. Dynamic documents can be used for debugging as well as editing. It should be possible, for example, to construct a dynamic document containing all sites where a particular global variable can be set. Dynamic documents can also be used to span different software artifacts. If the user cites a class, the appropriate dynamic document could include the OMT diagram for the class, its documentation, and the header file and method bodies that implement the class.

Our third objective was to maintain an open environment. One of the principal reasons that control integration has been successful is that it is easily adapted to work with existing tools and existing code. In order to remain compatible with existing tools, we needed to use the current file structure, maintaining individual source files for use by the compiler, debugger, and other tools. In order to support existing systems, we needed to be able to support multiple languages simultaneously. This includes both standard programming languages and preprocessed languages such as *lex* or *yacc* input. We also wanted to remain compatible with the existing UNIX programming environment so that our tools could be used in parallel with existing tools and so that new tools could be adapted incrementally. Finally, we wanted to make it easy to add tools, both our own tools and tools from different vendors.

2.0 Integration Mechanisms

In order to meet these requirements, we have developed a three part integration mechanism. The first part uses standard control integration. Here we are using the FIELD message server interconnected with Sun's ToolTalk message bus. The second part uses a simplified form of data integration we call *fragment integration*. This involves creating a simple database, identifying logical portions of files, and storing references to these portions as well as additional information to define links and store associated data. The third part uses a common editor framework that can support hyperlinks, display, and editing of a wide variety of software artifacts. Here we are using FrameMaker along with our own APIs. An overview of the environment showing how the components fit these categories is shown in Figure 1. The rectangular boxes represent the components of the Desert environment. The rounded boxes represent existing framework components, while the elliptical boxes represent existing tools or tool sets.

```
DEFINE
  FredGotoMsg =  [ FRED GOTO %1s %2d %3s %4s ];
  FredAddGotoMsg=[FRED ADD_GOTO %1s %2d %3s %4s ];

  FocusItemMsg = [ %s USERFOCUS %1s %2d %s %s %3s ];
  FormFocusMsg = [ FORM USERFOCUS %1s %2d ];
  FormErrorMsg = [ FORM ERROR %1s %2d %s ];
  FormWarningMsg = [ FORM WARNING %1s %2d %s ];

TOOL Fred
LEVEL User:
    WHEN FocusItemMsg(file,line,str) DO
      SEND FredGotoMsg(file,line,"Focus",str)
    WHEN FormFocusMsg(file,line) DO
      SEND FredGotoMsg(file,line,"Focus","*")
    WHEN FormErrorMsg(file,line) DO
      SEND FredAddGotoMsg(file,line,"Error","*")
    WHEN FormWarningMsg(file,line) DO
      SEND FredAddGotoMsg(file,line,"Warning","*")
END
```

FIGURE 2. Extract from Policy Message Mapping File

2.1 Control Integration

The use of control integration in the environment followed from its success in FIELD and the various commercial environments. Control integration based on selective broadcasting provides a simple foundation that allows diverse tools to communicate. It is easily extended by defining new messages for tools. Moreover, it is fast and can be readily adapted to a wide variety of tools either by doing minor modifications to those tools or by providing wrappers. In Desert, we use the FIELD message server (MSG) as the basis for our control integration [18].

Our use of control integration in Desert differs from its use in FIELD in two ways. First, we are using the FIELD policy tool as a mapping engine. Each tool defines its set of input and output messages locally. A policy file then defines mappings between tool output and tool input messages. An example of this is shown in Figure 2. This provides a very flexible framework where none of the tools needs to know the messages produced or consumed by any other tool. The second difference is the addition of a new

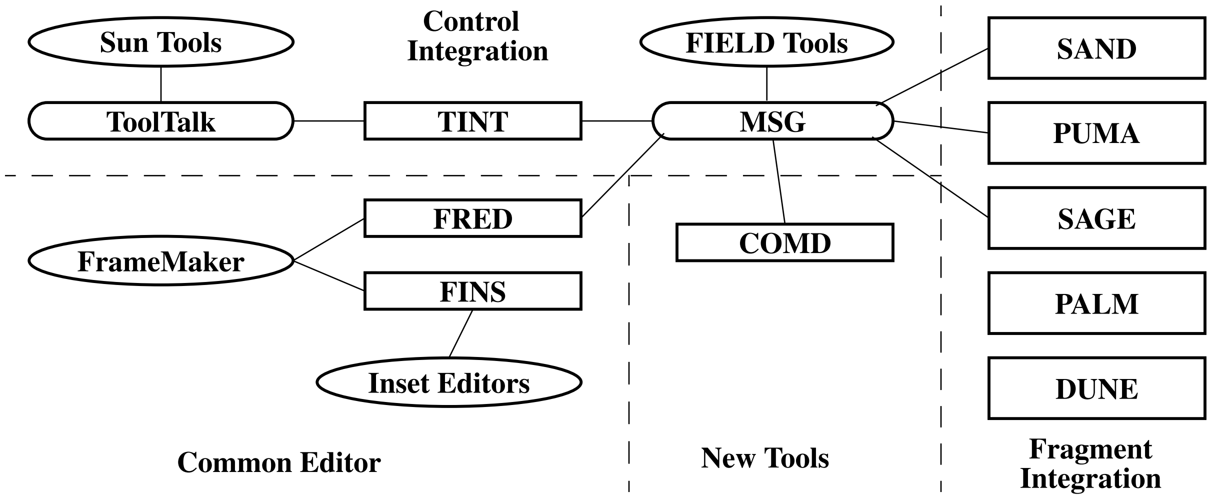

FIGURE 1. Overview of the Desert architecture. The architecture can be divided into 4 parts. The portion dealing with control integration includes the FIELD message server MSG and TINT to convert between ToolTalk and FIELD messages. The common editor portion includes the FrameMaker APIs FRED for program editing and FINS for inset editing of various software artifacts. The fragment integration portion includes the SAND fragment and cross reference database, the PUMA project manager,and the SAGE scanner. The PALM position and line manager and the DUNE interface provide SAND, PUMA, and SAGE functionality to other tools. Finally, the new tools portion of the architecture includes the context interface COMD.

tool, TINT, that serves as an interface between the FIELD message server and Sun's Tooltalk message server. This, combined with the mapping engine, allows our new environment to communicate directly with Sun's (or SGI's or any other vendor using ToolTalk) programming tools and eliminates the need for tool modifications or wrappers.

2.2 Fragment Integration

The second piece of the Desert integration mechanism provides services normally associated with data integration. Most approaches to data integration put as much information as possible into a central database to allow the various tools to share and reuse the information. We take the opposite approach, attempting to store the minimum amount of information needed to achieve our objectives. Our current system generates a database that about five times the size of the total source code, and hence is typically smaller than the object code with debugging information.

Our design here was based on several insights. We first noted that the basic program unit that the database needs to refer to is relatively large. While the traditional unit of a file is too large for our objectives, using the basic syntactic constructs of the language (such as using Diana for ADA [13] or as in ProCase [5]), stores more information than necessary and is too language specific. For most applications it is sufficient to break a program down into language-independent logical units such as files, functions, type definitions and variable declarations since these are the units the programmer will refer to. Other software artifacts can be broken down similarly. For example, an set of OMT diagrams can be broken down into the individual diagrams and a user interface design can be divided into the different windows and dialog boxes it provides. We call this logical unit a *fragment* and use it as the basic element of the database. Next we noted that in order to maintain openness we needed to preserve the original source files. Rather than effectively duplicating these source files in the database, we keep indirect pointers to each fragment in its original file. Finally, we noted that the other information that was needed in the central database was either information that was generated by one tool for use by another or was information that would be used to identify a set of fragments. Information sharing is done at the fragment level using property-value pairs where each tool can define a set of properties.

While there are many ways of identifying a set of fragments in the database, most of the applications we anticipate require use-definition information. To facilitate these applications, the fragment database also maintains enough data to reconstruct use-def chains on the fly by effectively maintaining the program's symbol table using semantic information garnered from the compiler and other sources. Here fragments are used to represent program scopes The database component of Desert is managed by two packages. The actual database is maintained in memory by the SAND package. It uses the message server to listen for query and update requests from other clients. It periodically checks all the files maintained by the database to see what has changed and uses the SAGE package to scan any modified files and update the database.

2.3 Common Editor

The other key objective in designing Desert was to provide a common editor for accessing software artifacts. We wanted this editor to be fully integrated with the environment so that users would feel they were editing the whole program rather than individual files. This implied the use of implicit and explicit hypertext links and live-links to special purpose editors. We also wanted to make full use of the power of modern displays by providing the user with a high-quality textual display such as recommended by Baecker and Marcus [1].

To meet these objectives we chose to build our editor on top of the commercial word processing system FrameMaker. FrameMaker is available on a wide variety of platforms. It has built-in hypertext capabilities as well as powerful editing operations. Moreover, it is user-extensible using the Frame Developer's Kit (FDK) [4]. This kit allows the definition of inset editors that support a simplified version of live-links and the definition of general purpose APIs. We use the FDK both to provide an inset interface, FINS, to external editors such as Paradigm+ for OMT diagrams and Builder Xcessory for user interfaces, and to provide an intelligent program editor, FRED, that makes use of the Desert integration facilities [20].

The editor makes extensive use of the capabilities of the environment. It uses control integration to interact with other tools, allowing other tools to specify a source position to be viewed in the editor and allowing the editor to initiate build, configuration, and debugging commands. It uses the database to support name lookup for both definitions and references over the whole system to allow the editor to support the editing of programs rather than of files, to provide implicit links based on program content, and to support incremental parsing on a keystroke basis. It uses the context manager to defined temporary contexts so that even files that are not part of a defined system can take advantage of Desert's capabilities.

3.0 Desert Concepts

Central to the design of the Desert environment was the detailed design of the fragment database, efficient identification of fragments and associated information, the concept of virtual fragment files as an approach to specialized documents, and the notion of a context. In this section we provide details on these.

3.1 The SAND Database

The fragment database provided by Desert serves three functions: it serves a repository of information about fragments; it provides the auxiliary data and query mechanisms needed to define and extract relationships among fragments and to identify related sets of fragments for creating virtual files; and it offers a repository of information about the source code that can be used by other tools such as the editor or visualization tools.

The actual database system is an in-memory relational query engine with object-oriented extensions, including object identifiers and method calls modeled on previous work on extensible database systems [12,16]. Because the size of the database is less than an order of magnitude larger than the size of the source code, keeping the whole database in memory is possible and allows us to simplify the database system while providing fast access to the underlying data. The database implements a SQL-like query language that is accessible through a message interface to other tools. In addition, it currently offers specialized message access for faster handling of more common queries such as name lookup for the editor.

Simplicity of implementation was central to our design of the database system. The system assumes that most queries can be handled quickly and there is little need to overlap query requests. As such, it processes one request at a time and does not deal with concurrency control and locking issues. The database itself can be rebuilt relatively fast (currently it takes under five minutes per megabyte of source code to completely rebuild the database), so that the database system does not need to worry about backup and recovery other than providing consistent caching of the data on disk. Finally, because almost all updates are handled internally to the database, there is little need for integrity and security management.

The database implementation currently provides the relations shown in Figure 3. In addition, it supports the notion of temporary lookup contexts for the editor. These are defined by specifying a base file and a set of include files. The database supports lookup operations based on a context for both the definition of a name and the uses of a name. These operations can be qualified by the name of a class defined in the context for looking up names defined in a member function or names used in an expression.

3.2 Updating the Database

While fast query access was important to the design of the database, the ability to quickly update the database as files changed was more important. Unlike conventional database systems, a database system providing data integration in a programming environment is updated as frequently or more so than it is queried. Each time a file is

File
- pathname
- date last modified
- source language
- compilation directory

Fragment
- fragment name
- CRC
- date last modified
- parent fragment name
- scope parent fragment
- fragment type
- start position
- end position

Use
- from fragment
- to fragment
- type of use

Definition
- name
- source fragment name
- object type
- scope type
- source line

Reference
- name
- source fragment name
- reference type
- source line
- read/write access flag
- local reference flag

Call
- from name
- to name
- source fragment name
- source line
- virtual call flag

Hierarchy
- from class name
- to class name
- source fragment name
- source line
- virtual flag
- friend flag

Member
- member name
- source fragment name
- source line
- type of member
- protection
- virtual flag
- inline flag
- static flag
- friend flag

Attribute
- attribute name
- redefinition scanners

Property
- fragment name
- attribute name
- value

FIGURE 3. Relations contained in the SAND database

edited or compiled, information in the database potentially needs to be modified.

SAND manages updates in three ways. First, at start-up it checks for any new files, tests if any files that it is maintaining have been modified, and updates its information accordingly. Next, while it is running it periodically (currently every 30 minutes) updates any modified files. Finally, it handles update requests through the message server either for an individual file (generated by the editor upon a save) or for the whole database (generated by user request from some tool).

Updating the SAND database is done by a set of scanners bundled together as SAGE. SAGE runs as a server process accepting messages requesting that a particular file be rescanned in a particular way. It currently provides source scanners for C and C++ source files, OMT diagram description files, UIL files, and VALLEY visualization description files [19]. These define the *file*, *fragment*, and *use* relations, identifying the fragments in the various files. In addition, SAGE provides a scanner for Sun's source browser data files. These run as if they were additional scanners for the corresponding source files, adding information generated by the compiler to define the *definition*, *reference*, *call*, *hierarchy*, and *member* relations.

The database was designed for efficient update on a file basis. All links in the database are stored indirectly. References to fragments are defined via fragment names rather than as actual links or pointers. This allows us to delete and reinsert all entries for a particular file without having to check the whole database to update link information for other files. Each tuple of each relation has tags indicating which file it came from and which scanner generated it. When a scanner is run, the database removes all previous tuples that were generated by that scanner for the particular file, and then runs the scanner to add new tuples.

3.3 Virtual Files

One of the objectives of the Desert environment was to allow the user to view a software system as a dynamic document rather than as a static set of files. To achieve this the environment uses virtual or *fragment files*. These are files consisting of a set of fragments extracted from their original source files. Each fragment is preceded by a brief header containing its source file, unique name and additional information for the user. In addition, the file itself contains a header describing how it was generated.

An example of such a file shown in the FRED editor is seen in Figure 4. The fragment header lines all begin with an at sign (@). The first five lines indicate that this is a fragment file with default language C++ constructed based on references to the identifier *name* in class *EMPLOYEE*. Two fragments are shown on this page. Each is preceded by four header lines which indicate which file it came from and the unique name of the fragment in that file.

The environment offers several facilities to support fragment files. First, a simple mechanism is provided to allow tools to easily build fragment files from a list of fragment names. This mechanism insures that duplicate fragments and fragments nested in other included fragments are ignored, and creates the file with all the appropriate headers and the fragment bodies. Second, it provides the logic for saving a fragment file after it has been edited. Here it uses the headers to isolate the various fragments from the fragment file. Then it checks the CRC (cyclic redundancy code [21]) for each isolated fragment against that of the original fragment to see if the fragment has actually been modified. If it has, it will replace the fragment in

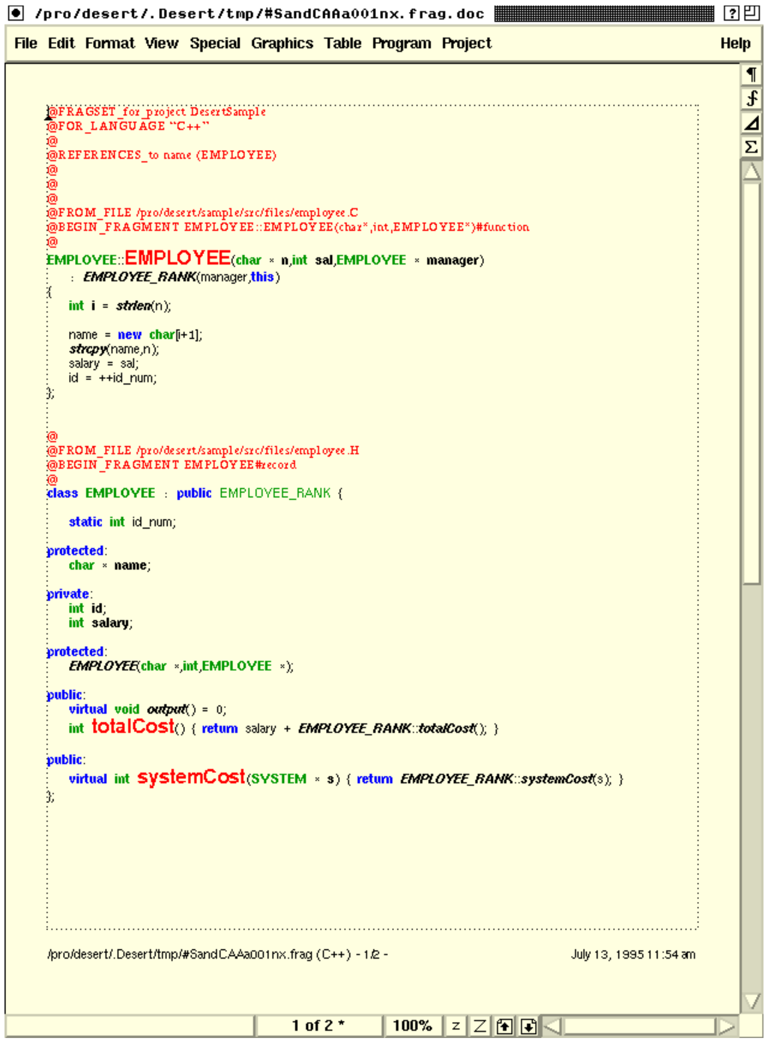

FIGURE 4. Sample fragment file

the original source file with the edited form, using an appropriate replacement mechanism based on the fragment type.

While we anticipate several mechanisms for generating fragment files in the full environment, the current implementation only allows the user to generate such files from the FRED editor through one of two mechanisms. The first allows the user to select a name in the source and construct a fragment file of all references to that name. The second, more general, mechanism creates a fragment file based on the set of gotos on the stack the editor maintains. The editor keeps a stack of file-line number pairs corresponding to locations of interest. Items can be added to this stack by requesting all definitions or references to a name or using the result of *grep*. Items are also automatically added for compilation errors and warnings if the FIELD tool *formview* is used to initiate the build. The user can edit the stack as well. For both of these cases, the editor first finds the smallest fragment corresponding to each file-line pair and then uses the mechanism provided by the environment to build the fragment file.

3.4 Contexts

The Desert framework has been designed to handle large-scale software engineering. This involved both storing information about the set of systems that correspond to a single project and offering common services to multiple users. Desert manages this through a context database.

Desert maintains information about projects in a context database. It supports three types of contexts. Global contexts are shared among a set of users. Local contexts are specific to a single user. Temporary contexts exist only while they are being used and are deleted afterward. Contexts may be defined relative to a base context. In this case they inherit the definitions of the base context and can provide any additional definitions of their own. This allows different contexts to represent different versions of a system.

Each context has a set of associated information including a unique context name, the location of the context's database, the type of context, a set of path names to use in the context, a set of path names to exclude, and a set of associations. Associations are mappings from files in the context to data. A variety of different associations are provided. The INCLUDE association allows the user to specify paths to be used to find include files while scanning. Other associations are provided to allow contexts to be integrated with both a version control and a configuration management system.

The PUMA package serves as the context database. It runs as an independent process and allows clients the ability to query and set information about the various contexts. It also allows the client to identify the context that is associated with a given source file for a given user.

4.0 Prototype Implementation

To test the various components of the system and to experiment with the concepts involved, we have built a prototype implementation and started using it for our own development. There are two current user tools, the FRED front end for editing and the context manager, COMD. In addition, we have implemented an initial version of the SAND database, SAGE scanners, and PUMA context manager. All these are supported by PALM for accessing fragments and DUNE utilities for accessing SAND, SAGE and PUMA. In this section we describe the problems that arose during this development and some of their solutions.

4.1 Fragment Scanning

While fragment scanning looked easy in principle, in practice there were several difficulties. Many of these were caused by the C preprocessor. In order to understand C or C++ files we had to expand macros since they are often used to define one or more functions and to define composite function names. In order to find all the macros we had to scan all the files included by the source file. Our scanner thus incorporates a simple implementation of the preprocessor which scans include files only to find macro definitions for later expansion. The first version of the scanner read each include file as it was needed. This turned out to take about 30% of the overall scanning time. We sped this up by caching macro definitions so that each include file needs to be read only once.

The other aspect of the preprocessor that complicates fragment scanning is conditional compilation. If we are interested in a particular instantiation of the system, we would know which conditions are applicable. However, because we are concerned with evolving the system in general, we assumed that all conditions can apply for some version of the system and thus we currently extract all the code independent of conditional compilation. Our experience has shown that this assumption is not generally correct. There are many conditions that are never applied (i.e. KERNEL in UNIX include files) to a system. Moreover, the user might want to define contexts where only certain conditions apply. We will be adding support for these situations in the scanner, the database, and the editor in the future.

Another difficulty arose in dealing with comments. The start and end positions of a fragment must take into account comments and spaces that are not generally considered part of a programming language. The general problem of determining which token a comment applies to is quite difficult as has been shown in the Mentor [6] and Sun's Clarity environments. We are dealing with a more limited case since we are only interested in comments for fragments. Our approach is twofold. First, the end position of a token is defined to include any blank spaces and comments up to the end of the line in which the token occurs. This associates end-of-line comments with the proper fragments. Second, we check for comment lines that precede the start of a fragment and associate these with the fragment if there are not too many blank lines separating that comment from the fragment. This handles block comments that precede function or type definitions.

4.2 The Editor Interface

The primary user interface to Desert thus far has been the API extensions we made to FrameMaker described in [20]. The editor interface has shown the power of the overall environment. We have had to augment our original design in only minor ways to provide services to the editor. The largest addition was the notion of a lookup context to the SAND database to support queries for a file that was new or modified since the database was last updated. The current implementation is fast enough so that the editor can query the database on each keystroke and the user will not notice any slowdown. Other enhancements included the ability to build fragment files from file-line number pairs, a search capability that used the fragment database to find the set of files relevant to the current context and then ran the UNIX grep command over those files, the ability to update a single file in the database, and temporary contexts so that a database existed for the current file.

The primary unresolved issues dealing with the editor interface involves fragment files. While it is convenient to generate fragment files from the editor, fragment files are generated purely from the database and hence do not take into account fragments that are currently being edited. It is

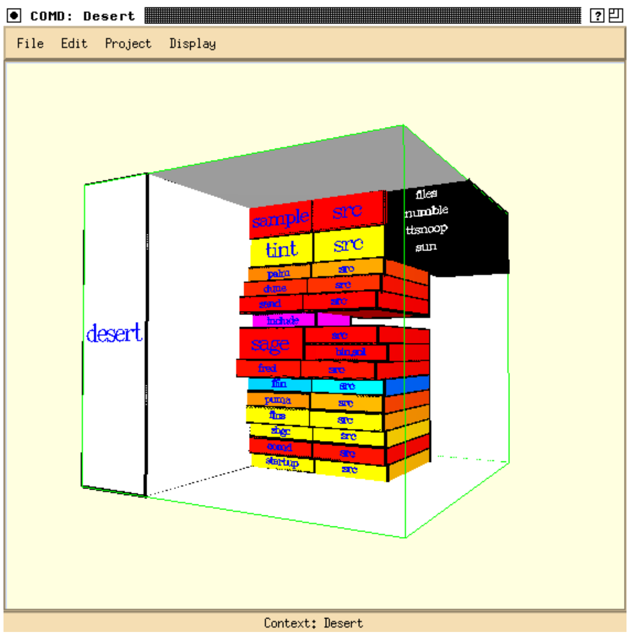

FIGURE 5. The COMD Interface

easy then for the user to create a fragment file that contains fragments from the file currently being edited and thus have multiple versions of those fragments being edited at once. This is a general problem that can occur whenever multiple fragment files exist. Another problem occurs when a fragment file is saved and the user has an editor open on the original file. Ideally, the fragment should be updated in the editor as well.

While these issues have not been serious yet, we anticipate that they will become more important. To manage them we are working on developing a means for locking fragments and for replacing fragments in an active editor rather than just in the original source file. Ideally, we would like to integrate this with a version control system.

4.3 The Context Interface

The COMD interface to Desert was originally conceived as a simple front end for context management, allowing the user to define and edit contexts as well as control updates. As shown in Figure 5 it provides a set of pull down menus for this purpose as well as a display area showing what is in the context. For the display, we are using the 3D facilities provided by the tools we developed for program visualization [19]. White nodes represent directories explicitly used in the context; black nodes represent directories explicitly excluded; otherwise a nodes color represents the date last modified of that node or any file in it. The height of a node represents the number of files. The display can also show the particular files that are included in the database.

As we gained experience with the environment, we have assigned more functionality to this interface. This interface provides a convenient location for selecting files to edit and for defining fragment files using either prespecified or user-defined queries. It provides a natural front end for configuration management and version control similar to the *formview* tool provided by FIELD or the POEM system [9]. It also offers a general framework for defining top-level visualizations for program understanding. While none of these has been implemented at this point, we have designed the framework for this interface to accommodate these.

5.0 Related Work

Integration in a programming environment today is usually viewed as a combination of control integration and data integration. Most commercial environment rely solely on control integration. PCTE [3] and evolving environments are emphasizing data integration as a needed facility.

Control integration, a technique that we pioneered for programming environments in the FIELD system [17], is very effective for integrating a concurrently running set of tools. It is the basis for most current commercial programming environments including HP's Softbench and Sun's Tooltalk. It provides a backbone for tool communication through a central message facility and broadcast messaging. It is relatively inexpensive and allows the use of existing tools with only minor modifications or through simple wrappers. Moreover, as we demonstrated in FIELD, it can be completely compatible with the use of existing tools. While control integration can make an independent set of tools look like a single unified framework, it does not provide complete integration, and does not store information over time. It also forces all the tools to have a common reference framework, typically a file name and line number, which limits its applicability. Control integration also only provides surface integration among the tools — it does not provide for any permanent shared data or for integrating multiple users over time and space.

Data integration combines tools by storing the program and information about the program in a database that is accessed by all the tools. Most proposed methods store annotated abstract syntax trees. Some more general systems extend this by using object-oriented databases to include information from other tools and phases of software development and relationships among the data. The database connects the various tools by allowing the tools to store their intermediate results for use by other tools. Fully utilizing a database provides a high degree of integration. Data integration has been used to some extent in the Ada programming support environments [13], in software engineering environments based on PCTE [3], and in commercial tools such as ProCase [5]. More recent systems along these lines include Centaur [2] and the Clarity environment at Sun Microsystems.

Unfortunately, practical data integration is difficult to achieve. The amount of data represented by intermediate results, especially for a large system, can be immense (i.e., gigabytes). It is difficult to agree on a common intermediate representation for tools for a single language, and even more difficult if the system has to handle a variety of different languages in a unified way. To achieve the potential of data integration, all existing programming tools have to be substantially changed or completely rewritten. The database that is required differs from most off-the-shelf systems in that it must be update-centric rather than query-centric. Finally, a database system capable of storing the amount of information required and providing the required performance and reliability is a large, complex piece of software, possibly larger than all the other tools in the environment combined.

An alternative to complete data integration has been to provide a program database as another tool in the environment. Interlisp's Masterscope package used an internal database [25]. Linton proposed using relational databases [10]. More recently, the FIELD environment provides a cross-reference database of program information that is used by a variety of tools in the environment [8], CIA and CIA++ represent environment-independent program databases for C and C++ respectively [7]. Sun's programming environment includes a similar tool, the source browser, that maintains its own database. Similarly, *cvstatic* from SGI uses either a compiler-based scanner or *cscope* from Bell Laboratories for fast generation of approximate semantic information. *SNiFF* also provides approximate information for a program database [24]. The use of an independent program database addresses some of the issues of data integration, but not all. They are limited to the source code and do not extend to other aspects of software development. They provide detailed information about variables, types, etc., but it is difficult for the systems to handle multiple languages cleanly. Finally, most of these systems require that the code compile in order to be included in the database since the scanners are either built into the compiler or are effectively full language parsers.

Another trend in programming environments today is to have an environment based on control integration with specialized data repositories. This is essentially a formalism of what current environments do. For example, FIELD uses control integration, but provides separate servers that offer cross reference data, profiling information, configuration and version management information, symbol table data, and run time tracing. Fragment integration is not an attempt to replace these specialized data repositories with a single one. Rather it is an attempt to provide a new repository that will hold the data needed to integrate tools.

Literate programming [14,15] is another approach that is related to ours. Here a the user creates a single file that contains documentation and source code intermixed and various tools exist to extract the source for the compiler or the documentation for TEX. This approach provides the user with a very readable program, but does not easily scale up for large systems or handle other aspects of software engineering. One of the benefits of our approach is that we should be able to simulate much of literate programming by choosing documentation and code fragments from their existing files, presenting them to the user in a single, editable form, and then extracting the fragments from the result and storing them in their original files.

Our approach to integration is based on an interface to the original source files that emulates data integration. The general concept of providing a wrapper for a file or system or of providing a view of an object has been around for a long time. It can be seen in current technology in the interfaces that are being developed for Mosaic. Here there exist virtual files at various sites that when accessed as a file, actually invoke a computation of some sort. Other related work can also be seen in databases that try to abstract information from files such as the Rufus system [23].

6.0 Experience and Conclusions

We have had some experience with the Desert environment. While the prototype is still too premature to release to even selective user communities, we have been using the system to develop itself for the last two months. The prototype involves about 60,000 lines (1.5M) of C++ code.

One of our early concerns with the environment was with performance. The performance of the various tools has been adequate. Database response has not been a problem, nor has the fact that we are using multiple processes communicating via the FIELD message server. The only serious performance problems involve loading and storing the database and opening a new file in the editor. The database for Desert is about 12 megabytes long and takes considerable time (i.e. a minute) to read in or store. To avoid the start-up time, the database system will continue to run for an hour after the last use. To avoid the store time, we currently store the database in background while we continue to process queries. The problem with editor start-up time involves the interaction of FrameMaker and our API and the inherent limitations of FrameMaker. We are hoping that the next version of FrameMaker which provides a tighter integration of an external API will alleviate these problems.

Other than performance issues, our experiences with the system have been mainly positive. The environment is capable of providing the services of data integration without the costs and in an open framework. Fragment files are generated quickly and are complete. Fragment editing does provide a convenient framework for many of the changes that are typically made in a set of source files.

We are continuing to work on the overall environment. Our current efforts are concentrated mainly in improvements to make the editor truly practical for a wide range of users and on integrating the current facilities with both version and configuration management. The current COMD interface provides a good starting point for a unified visual front end for version and configuration management. A fragment-based approach, however, requires that locking and versioning be done on arbitrary, possibly overlapping, sets of fragments rather than on files or directories as in current systems. We are investigating general models for version and configuration management that allow this and still are compatible with of a variety of existing tools. Finally, we are continuing to work on our interface to Tooltalk in order to more fully integrate the environment with programming tools from different sources.

7.0 Acknowledgments

Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225, by ONR grant N00014-91-J-4052, and by support from Sun Microsystems and NYNEX.

8.0 References

1. Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley (1990).

2. P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "CENTAUR: the system," *SIGPLAN Notices* Vol. **24**(2) pp. 14-24 (February 1989).

3. Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An overview of PCTE and PCTE+," *SIGPLAN Notices* Vol. **24**(2) pp. 248-257 (February 1989).

4. Frame Technology Corporation, *FDK Programmer's Guide*, Frame Technology Corporation (Octover 1993).

5. PROCASE Corporation, "SMARTsystem Technical Overview," PROCASE Corporation (1989).

6. Veronique Donzeau-Gouge, Gerard Heut, Gilles Kahn, and Bernard Lang, "Programming environments based on structured editors: the MENTOR Experience," in *Interactive Programming Environments*, ed. D. R. Barstow, H. E. Shrobe and E. Sandewall,McGraw-Hill, New York (1984).

7. Judith E. Grass and Yih-Farn Chen, "The C++ information abstractor," *Proceedings of the Second USENIX C++ Conference*, pp. 265-275 (April 1990).

8. Moises Lejter, Scott Meyers, and Steven P. Reiss, "Support for maintaining object-oriented programs," *IEEE Trans. on Software Engineering* Vol. **18**(12) pp. 1045-1052 (December 1992).

9. Yi-Jing Lin and Steven P. Reiss, "Configuration management in terms of modules," *Proc. 5th Intl. Workshop on Software Configuration Management*, (April 1995).

10. Mark A. Linton, "Implementing relational views of programs," *SIGPLAN Notices* Vol. **19**(5) pp. 132-140 (May 1984).

11. Scott Meyers, "Difficulties in integrating multiview development systems," *IEEE Software* Vol. **8**(1) pp. 50-57 (January 1991).

12. Gail Mitchell, "Extensible query processing in an object-oriented database," Brown University Computer Science Technical Report CS-93-16 (May 1993).

13. Robert Munck, Patricia Oberndorf, Erhard Ploedereder, and Richard Thall, "An overview of DOD_STD_1838A (proposed), the common APSE interface set, Revision A," *SIGPLAN Notices* Vol. **24**(2) pp. 235-247 (February 1989).

14. Norman Ramsey, "Literate programming tools need not be complex," Princeton University Department of Computer Science Research Report CS-TR-351-91 (October 1991).

15. N. Ramsey, "Literate programming: weaving a language-independent WEB," *CACM* Vol. **32**(9) pp. 1051-1055 (September 1989).

16. Steven P. Reiss, "*Eris*: the design and implementation of an experimental relational information system," Brown University (1983).

17. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. **7**(4) pp. 57-67 (July 1990).

18. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

19. Steven P. Reiss, "An engine for the 3D visualization of program information," *Journal of Visual Languages*, (To appear in 1995).

20. Steven P. Reiss, "Program Editing in a Software Development Environment," Brown U. Computer Science (1995).

21. R. Rivest, "The MD5 message-digest algorithm," MIT Laboratory for Computer Science and RSD Data Security, Inc. (April 1992).

22. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall (1991).

23. K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, "The Rufus system: information organization for semi-structured data," *Proc. 19th VLDB Conference*, pp. 1-12 (1993).

24. TakeFive Software, *SNiFF+ Version 1.0 Reference Guide*, TakeFive Software (1993).

25. Warren Teitelman, *Interlisp Reference Manual*, XEROX (1974).