The Desert Environment

Revised Paper

Steven P. Reiss[1]
Department of Computer Science
Brown University
Providence, RI 02912
401-863-7641, FAX: 401-863-7657
spr@cs.brown.edu

Abstract

The Desert software engineering environment is a suite of tools developed to enhance programmer productivity through increased tool integration. It introduces an inexpensive form of data integration to provide additional tool capabilities and information sharing among tools, uses a common editor to give high-quality semantic feedback and to integrate different types of software artifacts, and builds virtual files on demand to address specific tasks. All this is done in an open and extensible environment capable of handling large software systems.

1. Introduction

The Desert environment was developed over four years as an extension of the message-based integration techniques used in our previous work on the FIELD environment. Our goal in implementing a new environment was to show that new and powerful facilities can be combined inexpensively with current tools in an open framework. This was motivated by the dearth of environments that effectively utilize information about the program to help the programmer.

The contributions of Desert are in three main areas:

- *Global Control Integration*: Desert augments standard control integration pioneered in FIELD with a global message server that lets everyone in the project team work together using common program information. The server allows the various tools of the environment synchronize their views of a complex project with low computational cost.
- *Inexpensive Data Integration*: Desert introduces the notion of fragments along with specialized data stores to provide the benefits of data integration without the costs. This allows the programmer to view the software using virtual files containing only the portions relevant to the task at hand. It provides the environment developer with the capabilities of a project repository while maintaining source files and an open environment.

1. Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225, DARPA order D885, ONR grant N00014-91-J-4052, Sun Microsystems, and Digital Equipment Corporation. The author would also like to thank the various anonymous reviewers for their helpful and detailed suggestions.

- *Common Software Editor*: Desert uses a word processor rather than a text editor and integrates it into the environment. This gives the user a single editor with a high-quality display for all phases of software development and lets the system provide semantic feedback to show the user potential problems as the code is written. It provides the environment developer a high-quality editor with advanced features as a basis for tool integration.

In the remainder of this section we provide the historical basis for Desert, noting both the FIELD environment and our goals for a new environment, and then provide a brief overview of Desert. The bulk of the paper provides a detailed account of the various pieces of the environment, how they fit together, and what they have achieved. This is followed by a look at related work and a summary of our experiences with the environment.

1.1 The FIELD Environment

The FIELD programming environment was developed in the late 1980s and early 1990s to demonstrate the feasibility of integrating a wide variety of software tools in an inexpensive manner [42]. FIELD introduced the use of messages for combining tools, a technique now called control integration. It uses messages to integrate tools for program editing, debugging, browsing, visualization, and configuration management into a single environment that appeared seamless to the user.

Message-based control integration combines software development tools using a central message server. Each tool is augmented either internally or with a simple wrapper so that it can communicate with the message server. Integration is then achieved by having tools send messages to the message server and having the message server redirect these messages to other tools as appropriate. To make this scheme work efficiently and to ensure that the various tools need not be aware of one another, the system uses patterns to direct the messages and views message sending as broadcasting.

Messages in general have two forms. The first, a synchronous message representing a command from one tool to another, is used, for example, by the editor to set a breakpoint in the debugger. The second form is an asynchronous notification message. Whenever the debugger takes control, for example, it sends out such a message indicating the location of its current focus. Editors and browsers use this message to update their display accordingly.

When a tool starts, it registers a set of patterns with the message server that describe the messages the tool is interested in receiving. These include the command messages that the tool will handle and the informational messages that the tool wants to monitor. Tools send messages to the central message server in order to request information or action from another tool or to send out information that other tools might be interested in. The message server matches this incoming message against all the registered patterns and forwards it to the appropriate tools. If the message was synchronous, it gathers the first non-null reply from any of the accepting tools and sends it back to the original sender.

This approach to integration has proven very powerful. When combined with an editor that uses annotations to interact with other tools (e.g. using a breakpoint annotation both to create and to show debugger breakpoints), a cross-reference database, a sophisticated wrapper for the system debugger, a graphical wrapper for configuration tools including *make* and *rcs*, a variety of visualization tools, and a common control panel, it yielded an environment that appeared to the programmer as a single entity rather than a diverse set of tools. This improved programmer productivity by simplifying the use of tools for the novice programmer and providing better, more intuitive user interfaces for existing textual tools.

In addition to message passing, FIELD was one of the first environments to make extensive use of an external database of program information. It included tools for automatically gathering and maintaining cross-reference information (effectively an extended symbol table) for the current executable. This information was used mainly by the various visualizations that FIELD provided, notably a call graph viewer and a class browser, by search tools, and in extended editor commands such as “go to the definition of what I’m pointing at”.

1.2 Our Goals for a New Environment

While FIELD provided a high degree of integration, we felt that more could and should be done. We wanted to focus on two particular aspects: *providing quick and easy access to data about the system being developed* and *providing a common editor for all phases of software development*.

Most attempts at providing access to program information have focused on providing a global repository of program information. While these approaches do provide a large amount of information, they have had serious problems with scalability, handling multiple languages simultaneously, handling multiple users, openness, and maintaining consistency with the actual program. In Desert, we focused on making only the basic information available without the overhead and problems of such repositories. In determining what information was required, we concentrated on feedback while editing and on the ability to provide virtual, editable views of the software, for example showing a function and all its call sites in a single display.

A second area where we saw room for improvement was in the diverse set of editors currently used for programming. Different editors are used for writing documentation, writing code, drawing design diagrams, specifying a system model, etc. Moreover, standard text editors do not do a good job of giving the programmer highly readable source code. We wanted to offer an environment that was centered around a single editor capable of handling a wide variety of programming tasks. Moreover, we felt it important that the editor provide the programmer with program text that was as readable as possible.

We also saw it was important to provide these facilities in an open and extensible framework. We wanted an environment that was compatible with existing tools and techniques so that programmers could migrate to the environment in stages and so that the environment could easily be used for existing systems. We wanted an environment that could handle new tools and new applications of existing tools. We

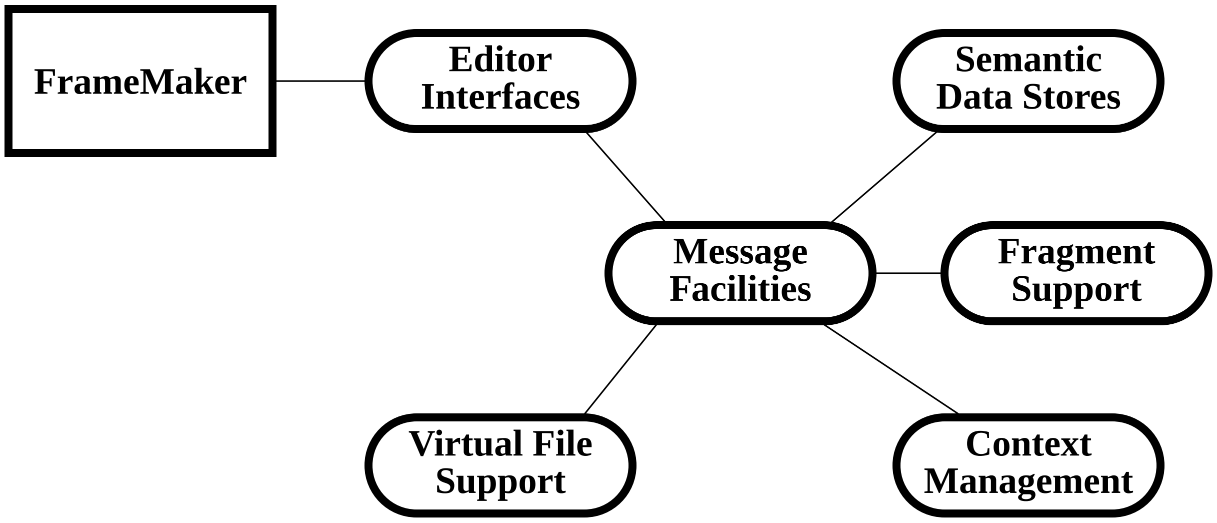


FIGURE 1. Overview of the Desert Environment.

wanted an environment that could scale to handle large-scale software projects (over a million lines of code in its prototype version, much more if the ideas were applied to a production environment) so as to demonstrate that any techniques that we developed were practical. We also wanted an environment that was as simple as possible to develop using existing tools, and was easy to implement or replicate.

1.3 An Overview of the Desert Environment

The Desert environment was developed with these goals in mind. In order to accomplish the goals, we had to combine several ideas, some of them new, other ones tried before in different, generally limited, contexts.

A top-level view of the Desert environment is shown in Figure 1. At the heart of the Desert environment and the center of this diagram are its integration facilities. Desert extends control integration from the FIELD environment with powerful mapping facilities that make it easier to reconfigure the environment dynamically and ensure that tools are totally independent. On top of this, Desert introduces an inexpensive form of data integration which we call *fragment integration* whereby logical units (*fragments*) of the original software artifacts are identified and references to them kept in a data store. Here, the primary storage is still the original software artifacts and not the data store, but tools can rapidly access and associate information with the logical units through the store. These ideas are discussed in Section 2.

Many of the benefits of data integration rely on a common repository of derived information. Desert replaces this repository with a collection of specialized data stores. The environment keeps the data stores consistent and up to date using non-intrusive techniques. It offers tools fast and easy access to the data through standard and specialized interfaces. This is described in Section 3.

One of our primary goals was to have the environment offer dynamic editable views of the software to simplify editing and program understanding. Desert provides such views through a concept we call *virtual files*. A virtual file combines fragments gathered from multiple locations in a single editable file. Desert supports creating, storing and locking such files. Section 4 looks at the mechanisms Desert provides here.

Desktop publishing technology has improved significantly over the past decade, yielding a variety of systems that offer WYSIWYG displays of both text and graphics and provide the facilities needed to integrate specialized editors with a common display. Desert exploits this technology by showing how a modern word processor can be extended to handle the tasks involved with software development. The principal extension offers an interface for coding using the data stores to provide feedback and high-quality text formatting. Additional extensions offer interfaces to specialized graphical editors for design and user-interface development, tools to help in design and documentation, and preliminary facilities for cooperative development. These are detailed in Section 5.

The specialized data stores, editing facilities, and integration mechanisms are all based on the notion that the programmer is working on a project and the underlying environment must be aware of what that project is, what it contains, how it is shared, and how it is developed. Desert maintains and offers to the rest of the environment a suitable system model that identifies projects and their properties using the notion of contexts. This is described in Section 6.

2. Basic Integration Mechanisms in Desert

Integration in software development environments can occur along multiple dimensions. Schefstrom and van den Broek [46] identify three dimensions: control, data, and user interface or presentation. Desert uses an extended version of FIELD's message-passing framework to provide robust control integration. It uses a new concept called *fragment integration* that offers a low-cost alternative to traditional data integration techniques. These serve as the basis for control and data integration and are described in this section. Desert offers broader data integration through specialized data stores described in Section 3. Finally, Desert provides for presentation integration through its emphasis on the use of the common editor described in Section 5.

2.1 Control Integration

Desert attempts to enhance the FIELD approach to message-based control integration to make it easier to use and extend it to large-scale software development.

Desert's first enhancement is to integrate and use a message mapping engine to make adding new tools and customizing the environment easier. Desert uses mappings to define how messages are used so that no tools needs to know the messages produced or consumed by any other tool and so that the interaction of the tools could be individualized.

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

   MscanErrorMsg = [ MSCAN ERROR %1s %2d %3s ];
   MscanWarningMsg = [ MSCAN WARNING %1s %2d %3s ];
   MscanBeginMsg = [ MSCAN BEGIN %1s ];
   MscanEndMsg = [ MSCAN END %1d %2s ];

TOOL Fred
LEVEL User:
      WHEN MscanErrorMsg(file,line,msg) DO
         SEND FredAddGotoMsg(file,line,”Error”,”*”)
      WHEN MscanWarningMsg(file,line,msg) DO
         SEND FredAddGotoMsg(file,line,”Warning”,”*”)
      WHEN MscanBeginMsg(wd) DO
         SEND FredClearAllMsg(“Error”,wd)
         SEND FredClearAllMsg(“Warning”,wd)
END
```

FIGURE 2. Extract from Policy Message Mapping File for FRED editor. This specifies, for example, that when a MscanErrorMessage is sent (by the build tool), the system should send out a FredAddGotoMsg causing the editor to go to that line, and that the MscanBeginMsg clears all previous error and warning messages.

This message mapping engine, MSGMAP, was based on ideas in the Forest system [17] and the related notion of mediators and events [50], and was partially implemented (although not used) in the FIELD system as the policy tool. MSGMAP can modify, resend, or hide any incoming message. It can take an arbitrary message sent out by one tool and transform it into a set of messages that serve as input for other tools. Its actions are determined by a textual resource file, so the tool can easily be customized for a particular work group or even a particular user. An example of such a resource file is shown in Figure 2.

A second Desert enhancement is designed to allow Desert to interact with existing tools. Desert adds a new tool, TINT, that serves as an interface between the Desert message server and a Tooltalk message server [51]. This tool demonstrates the feasibility of managing multiple message servers where necessary.

The final Desert control integration enhancement is designed to handle multiple-person development within the environment. Desert uses two message servers whereas FIELD only used one. In Desert, one of the message servers, like the FIELD server, connects the local tools to form an integrated framework. The other message server operates globally, providing common access for all users to the environment facilities. This is used in various ways. Projects in Desert can be defined either globally or locally. Global projects are supported with shared data stores that automatically reflect the changes of each use. These stores are accessed and managed through the global message server. This server also provides the system editor with mechanisms for fine-grain locking and cooperative editing. Moreover, it provides the hooks needed for a wide variety of cooperative support and notification tools.

2.2 A Basis for Data Integration

Data integration is used in software engineering environments to offer such facilities as:

- Semantic feedback while editing, compiling or debugging;
- Extensive data sharing among tools;
- Tracability of changes;
- Linkages among the data items;
- Query-based access to system information;
- Multiple views of the stored data; and
- Support for cooperative development.

These have typically been achieved either through a program repository or through shared files. Both of these approaches have their advantages and disadvantages and Desert attempts to take a practical middle ground.

In a repository-based environment all data about the system, typically to the level of abstract syntax trees, is kept in a database that the various tools can access. While such environments can in theory provide all the above facilities, in practice they have proven difficult to build and use. The largest problems, as we have mentioned, are with scalability, handling multiple languages in a single project, handling multiple users, openness, and consistency maintenance. In essence, this approach has generally proven too cumbersome and inefficient to be practical for large systems.

The alternative approach is to maintain individual files and to have tools share information by sharing these files as is typically done in UNIX. The disadvantages here are that the above facilities are often difficult or impossible to implement, that much of the work involved, especially program analysis, has to be done repeatedly by each tool, and that it is difficult to maintain consistency of the various files as the system is being developed. Despite these, files have significant advantages. Using individual files lets one develop a simple, open and scalable system. Files are relatively inexpensive with today's operating systems since the mechanisms for storing and accessing them are well developed and highly optimized. They also have the advantage of providing the reader of the software with the logical organization created by the original programmer. For these reasons, most current programming environments are still file-based.

Desert's approach to data integration is a compromise between using files and using a repository. Desert maintains the original files and uses them as the primary source of data; this provides the benefits of files. At the same time, Desert automatically extracts information from the files and saves it in specialized data stores, thus gaining many of the advantages of a repository-based environment. Further benefits are achieved by dividing each file into logical chunks called *fragments* and creating a data store of references to these fragments.

2.2.1 Fragments

A *fragment* in Desert is a section of a file that should be considered a logical unit by the programmer. The basis for fragments is the set of documents or artifacts that comprise a software system, such as source code files, documentation, requirements and specification descriptions, and design documents. These artifacts may be stored as actual files or may (like design diagrams in many of today's CASE tools) actually be sets of items stored in a database.

The primary motivation for fragments is that they can serve as the basis for better data integration. They represent the level at which associations between different program units and tool-specific information are maintained. They also represent the context to be presented to the programmer when displaying such links or using such information. A fragment-based representation is a compromise between a repository-based representation that works at the low level of an abstract syntax tree and a file-based representation that works at too high a level.

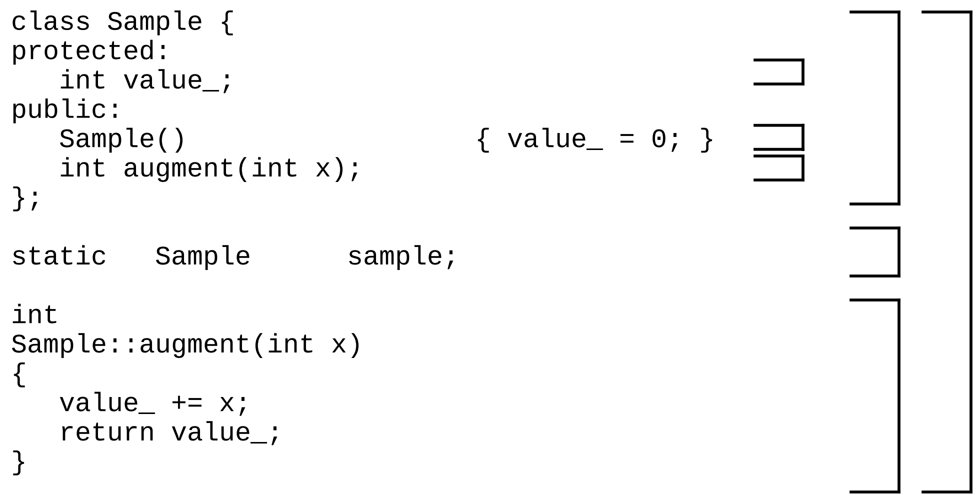
While Desert makes no specific assumptions about what is in a fragment, we have developed guidelines for defining fragments for different types of artifacts. Fragments should first and foremost represent a context that can be shown to the user as a meaningful unit. This generally implies so that a fragment should represent a logical portion of the artifact at a high enough level that additional context for that portion is not essential to its understanding. For example, a statement in a program is probably too low-level for a fragment, but the function containing that statement would be okay. If fragments are at too fine a level, the size of the database required to hold them, the time it takes to update and track them, and the complexity of analyses based on them will all be too large to be practical for large systems. Ensuring that a fragment represents a logical context reduces the number of potential fragments to a manageable number, even for a large software project.

Second, fragments should represent concrete concepts such as files, classes or dialog boxes. This provides a logical context for using fragments to share information among tools and offers a firmer basis for presenting the fragment to the user. Fragments that are concrete concepts provide a natural basis for a tool to associate information for other tools. They also provide a meaningful basis for establishing relationships such as the compilation dependencies inherent in a system model.

In the same way, a third guideline is that each file should itself be a fragment. Files are concrete entities that need to be represented within the environment. Having a single top-level fragment for each file lets fragments be used as files or vice versa, allowing, for example, file-based locking in conjunction with fragment-based locking.

Finally, fragments should represent hierarchical portions of files. While fragments can be properly nested in other fragments, two fragments at the same level of hierarchy should never overlap one another. This ensures that fragment-based locking works, allows the use of hierarchy, and ensures that a single location in a file corresponds to a single lowest-level fragment. This makes the implementation of fragments easy and logically consistent throughout the system.

With these guidelines, fragments can be created for a wide variety of documents. Fragments for a C++ program include files, function declarations, class definitions,



```
class Sample {
protected:
   int value_;
public:
   Sample()                 { value_ = 0; }
   int augment(int x);
};

static   Sample      sample;

int
Sample::augment(int x)
{
   value_ += x;
   return value_;
}
```

FIGURE 3. Sample C++ code fragment with brackets showing the different fragments.

and top-level variable, member, and type declarations. For example, Figure 3 shows a simple C++ file and the associated fragments. Fragments for documentation include title, sections, and subsections. Fragments for an OMT diagram represent the different sheets and, within each sheet, the various objects. Fragments for a user interface diagram represent the different top-level windows.

Desert creates and automatically maintains a data store of information about the fragments in a project. For each fragment, the store includes a source location specifying the start and end position in its source file. In addition, it includes the fragment's parent, a fragment type characterizing both the type of file it came from and the type of information it represents, a name that is relatively unique (as discussed below), a hash value of the contents of the fragment (computed using cryptographic techniques [45]) so that changes to the content can be detected, and a set of name-value associations that let arbitrary data be stored with the fragment.

In order to simplify updating and maintaining the data store, relationships among fragments are kept indirectly using the fragment name. Our motivation here was not only to let the data store be updated without having to maintain links, but also to let fragments be tracked as functions and other information is moved between files. This requires a consistent strategy for naming fragments that ensures that names are unique and based on the content of the fragment. We use a scheme similar to C++ name mangling [30] augmented to distinguish what would otherwise be duplicates within a file. We allow duplicate names in separate files and ensure unique references by always requesting fragments using a name-file pair.

2.2.2 Fragment Data Store

Desert's implementation of fragment integration has two parts: a simple data store of the fragment information and a set of scanners to identify fragments in different types of software artifacts. The data store is meant to contain the necessary information about the fragments and to let tools associate information with a fragment for

```
File
   pathname
   date last modified
   source language
   base directory

Fragment
   file name
   fragment name
   CRC checksum of contents
   date last modified
   parent fragment name
   fragment type
   start position
   end position

Include
   from file
   to file

Attribute
   attribute name
   redefinition scanners
   pattern describing legal values

Property
   fragment name
   attribute name
   value
```

FIGURE 4. Relations and fields of the fragment database.

their own or other tool's use. It is implemented as one of a set of specialized relational databases; the other databases and their common implementation are described in Section 3. The scanners are designed to allow the data store to be updated quickly, accurately and unobtrusively.

The fragment data store is a relational database consists of two relations that maintain the data store and three relations that actually store the information associated with each fragment, as shown in Figure 4. The *File* and *Include* relations determine what files need to be rescanned when the data store is updated by looking at which files have changed and which files are dependent on them. Dependencies here, while typically oriented toward the use of header files in programming, are actually generalized for just this updating function. The *Fragment* relation stores the information that is associated with each fragment. The start and end position here can be interpreted differently depending on the fragment and file types, thus allowing for fragments that are actually stored in containers other than simple files. Finally, the *Attribute* and *Property* relations let arbitrary strings be associated with a fragment for information sharing.

2.2.3 Fragment Scanning

In order to make fragment integration work, the system must be able to find and track fragments as the system evolves. This is done using a set of scanners that are designed to be as simple and fast as possible and are not full parsers for the corresponding languages. For example, our C and C++ scanners pay attention to block comments and blank lines as well as the syntax for functions and declarations. They totally ignore the syntax of statements and expressions. The scanners are also designed to handle incorrect and incomplete files in a logical way. When programmers are actively working on the system, files are often left in an intermediate, uncompilable state. In this way our scanners are similar to the various partial scanners that have been developed for quickly obtaining semantic information such as SGI's *cvstatic* or *SNiFF* [48].

Our scanners differ in that they must take comments into account. The scanners need to report to the data store the start and end positions of each fragment. Comments appearing in the source files should be associated with the correct fragment even though they may precede or follow its text. While the general problem of determining the token to which a comment applies is quite difficult, here we need only determine whether a comment that immediately precedes or follows a syntactical fragment should be considered part of it or not. Our approach has two parts. 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 associate any comment lines (lines that contain only a comment) that precede the start of a fragment with the fragment if there are fewer than K blank lines separating that comment from the fragment, where K is a user-definable parameter. This finds block comments that precede function or type definitions and associates them with the correct fragment.

While we developed a common framework for scanning source files, scanners for other document types have been written separately. This has not been a problem since most of these scanners are relatively simple (less than five-hundred lines of code) and the problems that arise are much less complex than with source files. For example, we have developed scanners for OMT files from *xomt*, database files from Cadre's Case tools, UIL files for user interfaces, and XD files from Sun's *visu*, We would expect that as the Desert approach is extended to include other areas, such as specifications or documentation, that more sophisticated scanners might be desired.

2.3 Evaluation

The control integration facilities provided by Desert are a natural extension of the successful mechanisms offered in FIELD. They have made it easy for the twenty or so tools currently composing the environment to interact with each other without imposing significant overhead or coding restrictions on the tools. Moreover, they have made it easy to add new tools.

The use of fragments for data integration, however, is more contentious. Fragments seem to be an appropriate mechanism around which to organize the data in a programming environment. They are a compromise between the low-level details that most repository-oriented environments store and the high-level notion of files. They form a good basis for presenting information to the programmer and should be useful for information storage and sharing between tools. At the same time, it is not always clear that the extra costs of this mechanism are needed nor that our approach to fragments is the right one.

Evaluating the worth of fragments in an environment requires that they be used to their fullest extent. This has not yet been done in Desert. The current set of tools make only limited use of the ability to save information with fragments. This is probably because the current selection of tools in the environment is limited, but we need more experience with the additional tools to see if fragments are a reasonable mechanism for data sharing and storage. Our presentation methods based on virtual files described in Section 4 are incomplete and we have not yet used fragments as a basis for system modeling or building. This means that much of the promise and opportunity we saw for fragments has been unfulfilled. Still, based on the advantages that can be gleamed even from the limited use we make of them, we feel that this approach is a strong step in the right direction. Note that some more recent environments such as Visual Age C++ [33] and CMU's Sheets [49] are using what can easily be seen as a fragment-based approach.

Our approach to fragments involves defining them, storing them, and scanning for them. Our guidelines for defining fragments represent a starting point. Our experiences show that a programming environment needs to work with units that are smaller than files and larger than syntactic constructs for presentation, editing, compilation, and information sharing. We have defined fragments based primarily on their use in source code. This approach has worked well for source-related details, but has not been tested extensively for other software artifacts. A better evaluation of this work will require understanding a broader range of the applications of fragments, especially for large system development.

Our fragment data store is quite fast both for query and update. However, we have found that the current environment does not need to access it very often. Most of its use comes from the creation of virtual files by extracting a logically related set of fragments, as discussed in Section 4. Here the data store is used mainly to find the proper fragment to display based on a location in a source file; it is not being asked to use fragment properties or other information in the date store. The other major use we envisioned early on was to provide an intelligent system-building tool that would determine when recompilation was needed on a fragment rather than a file basis. While the fragment data store has much of the information needed for this task, it proved much harder than anticipated to define precisely and then determine what constituted a compilation dependency between two files, and we have not yet implemented such a tool.

Finally, our work on fragment scanning has shown that scanners can be developed that are fast, unobtrusive, and can handle incorrect files as needed in an active development environment. Our approach to comment management in the scanners has worked very well for a limited set of examples. We have found few if any problems in the scanners' analysis of our own code or of a large portion of library code. However, it is quite likely that other coding styles and formats would confuse the current scanners. A more detailed analysis that considers a wide range of different coding styles is needed.

3. Data Stores

Many of the benefits of data integration come from making information derived from the original files available to the various tools. This makes a suitable database an essential part of a software engineering environment. To be practical, such a data store must update itself with a minimum of overhead and user intervention and must provide query mechanisms that are fast enough to meet response expectations (i.e. keystrokes in the editor) and general enough to address the needs of a wide variety of tools.

3.1 Specialized Data Stores

Desert achieves these goals using multiple data stores. It divides the program information into multiple categories and builds for each a specialized data store that is tuned to the particular needs of the data it stores. These specialized data stores are relatively small and cheap while, at the same time, offering superior performance. The current ones include one for fragment data and another for semantic data. Both of these are language-independent and currently handle C, C++, and Java source files, OMT diagrams, user interface resource files, and some documentation. Additional stores are envisioned to handle performance data, run time trace data, and to handle configuration data for both system building and version management.

These data stores are supported by a common framework in order to make it simple to add new stores and minimize the effort involved in implementing and updating each. The framework, a set of C++ classes that are easily inherited and customized, provides a relational database for each data store. It includes the code needed for interacting with the message servers and scanners, a query interface that supports SQL, storage and retrieval mechanisms for relations, and a general-purpose optimizing query engine. Creating a new data store then involves specializing two of the framework classes, one for defining the server and one for defining the relational database. The specializations needed for a basic implementation are quite simple and can be done easily in a day or less.

The framework also provides a generic interface for defining the relations in the data store. A new relation is added to a store by specializing an instance of this interface, defining the fields of the relation, and providing methods to read and write a tuple of the relation. This makes adding or modifying relations quite simple and has let us augment the relations as we added other data sources or needed additional information. All relations are kept in memory for processing and saved and loaded on disk only when needed and the framework supports indices for higher performance. All this ensures that the data stores can be readily adapted to new languages and changing data demands.

A relational approach was used here because most of the relevant data was already present in a tuple-based framework, because it is simple to implement and general enough to handle the variety of data we need to store, and because there are efficient and standard query languages available. Where we need to go beyond the relational model (for example in doing scoped name lookup), we defined specialized operations.

The overall architecture of the Desert data store implementation is shown in Figure 5. The two ovals on the left, PALM and DUNE, represent the library interface to the stores provided to all tools. These communicate to the actual data stores using the message server MSG. These are shown as rectangles to indicate that they are separate processes.

The actual data store code is broken into three parts. SAND provides the generic framework. The two data stores currently implemented, the fragment data store described in the previous section, FRAG, and the semantic or cross-reference data

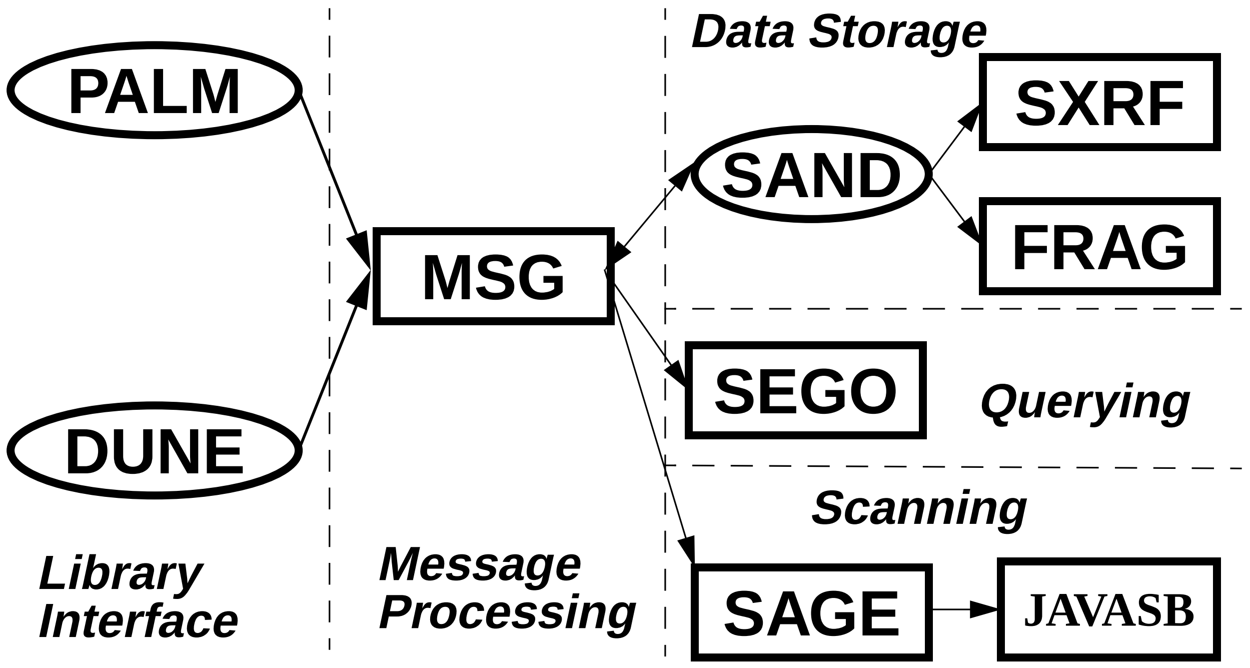


FIGURE 5. The basic architecture of the Desert data stores.

store, SXRF, are built on top of this framework. The SXRF store keeps track of symbol table information such as scopes, definitions, the class hierarchy, and access information about class members. It also contains cross-reference information such as a static call graph and all references to names. The complete set of relations and fields in the current data store is shown in Figure 6.

The next piece of the framework is SEGO, a common interface to the various data stores that takes generic queries and redirects them to the appropriate store, combining the results as needed. Currently SEGO only handles the limited set of queries needed by the Desert search engine CAMEL described in Section 4. It is essentially a placeholder for a true federated interface.

The final component, SAGE, contains the code for scanning software artifacts in order to obtain the data store information. It consists of both generic code and specific implementations for scanning the current set of support files. Some scanning, such as for Java programs, is done by independent scanners such as JAVASB.

3.2 Collecting Program Information

For these data stores to be a successful part of the environment, they must be maintained unobtrusively so that the data is available without the programmer having to specifically request it or to wait for it to be accumulated or brought up to date. This led to a variety of design decisions concerning the appropriate implementation strategy. The first of these involved generating the necessary data.

Information for the various data stores can be gathered from a variety of sources. As noted in the previous section, fragment information is obtained by scanning source

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

Include
   from fragment
   to fragment

Scope
   scope name
   parent scope name
   type of scope
   file name
   start line of scope
   end line of scope

Definition
   identifier name
   scope name
   type of object being defined
   type of scope
   file name
   line number of definition

Reference
   identifier name
   scope name
   type of object if known
   file name
   line number
   read/write/definition flag
   local/global flag

Call
   from routine
   to routine
   file name
   line number
   virtual flag

Hierarchy
   from class
   to class
   file name
   line number
   virtual flag
   friend flag

Member
   member name
   scope name
   file name
   line number
   type of member
   protection of member
   virtual flag
   inline flag
   static flag
   friend flag
```

FIGURE 6. Relations in the semantic cross-reference database SXRF.

files using a specialized parser that identifies fragments and can deal with comments, preprocessors, and errors. Semantic information for the SXRF data store is obtained in part in a similar way, using specialized scanners. However, such information is now typically available from the compiler. Desert, to save time and effort, takes advantage of this compiler-generated information whenever possible by reading the compiler-generated data files rather than the original sources. This turned out to be considerably faster

The various scanners are all embedded in the SAGE server. This is a separate process that handles scan requests through the message bus. The server handles requests by scanning the appropriate artifact and generating files that can be loaded directly into the data stores. It turned out to be more efficient and faster to implement this as a service rather than to spawn separate scanners for each file. The actual implementation allows multiple SAGE servers for handling different types of requests. This lets us process multiple requests in parallel and provides increased performance when updating multiple data stores.

Our current implementation of SAGE is quite efficient: most files are scanned in a second or less, and most of the time is spent on reading the files on input and writing the standard data files as output. When updating a large number of files, SAGE typically runs faster than the requesting system, so that the bottleneck is in integrating the information into the data stores, not in generating the information.

The only drawback to implementing SAGE as a service is that the code for scanning a new artifact type generally must be written as part of the server. To facilitate this, the server provides an interface class that can be redefined for each artifact type. The scanner for the artifact can either be written from scratch or make use of the current scanning facilities. We have found that, except for source code files, most artifacts are generally quite easy to parse for either fragments or the limited semantic information they contain. Adding a scanner for a new artifact type generally takes less than a day of effort.

3.3 Updating the Data Stores

The other part of the implementation strategy aimed at making the implementation of the data stores unobtrusive involved making update as fast and efficient as possible. This was achieved by doing all updates in a "batch" mode at the file level. The SAND framework provides the logic for updating the data stores as needed. When the framework detects that a given file has changed, it first determines which scanners for that file need to be rerun. Then, for each scanner, it removes all tuples that were derived for the file by the scanner, runs the scanner to generate a new data file, then reads this data file and adds the resultant information to the data store. All tuples to be removed for a set of files are actually just flagged for removal and the whole data store is cleaned up only once.

To make this update strategy work, the data stores must not contain any links between data elements that arise from separate files. This is necessary to let the framework remove and add tuples locally without having to worry about global consequences. Where links are needed (i.e., for an include file or the containing scope of a definition), they are computed dynamically from names or other information in the data store and cached for efficiency. The drawback of this approach is that it allows dangling links, for example a reference to an include file that is no longer in the data store. While the user of the data store should take the possibility of dangling links into account, these actually occur quite rarely in practice. In almost any case where references in a file change, the files referring to those references are either explicitly changed or at least recompiled or reprocessed (since the base file changed) so that their information is also updated. Other than very contrived situations, we have not seen any dangling links nor had them adversely affect any of the tools.

A final step in making the updates unobtrusive is to make them occur automatically. The framework provides the common logic for determining when a file should be scanned and updated. It keeps track of all the files in a project and can search for additional files created since the last update. It keeps track of the last update time and periodically (every 15 minutes) checks for changes and updates the data store accordingly. It also handles requests from other tools to update the whole data store or the data store relative to particular file. The Desert editor automatically sends a message to the data store requesting a specific-file update whenever the corresponding file is saved. The current front end to UNIX *make* sends a similar message after a successful compilation.

3.4 Query Mechanisms

In addition to efficient updating, the data stores must present an fast and general purpose query interface to the various tools, letting tools request information as needed without significant delays. The SAND interface provides for this in two ways: a general SQL-based query front end and a separate interface for special-purpose queries.

SAND provides a query evaluator based on an underlying set of relational operators along with a general-purpose query optimizer. The optimizer uses techniques developed for relational databases and provides an extensible set of operators [31,36]. In addition, a common front end takes SQL as input and translates it into operators for the evaluation engine, so that tools can send SQL queries to any of the data stores.

SAND also supports a message-based interface for requests that are specific to a particular data store. This is used for queries with high performance requirements that would be too slow through the standard mechanism. The fragment data store uses these commands to find the fragment corresponding to a given line in a given file. The semantic data store uses them to find the external definitions corresponding to a reference of a name in a given file and to find all references to these definitions.

3.5 Evaluation

The maintenance of the various data stores has proven quite effective. As long as the environment is used on a project on a regular basis, the automatic updating is unobtrusive and fast. This leads us to conclude that it is possible for a programming environment based on files (or fragments) to actively maintain a program data store that can be readily used by the various tools.

The largest project we have used the data stores for is Desert itself. Desert consists of about 250,000 lines of C++, or about five megabytes of source. The fragment data store for Desert occupies about six megabytes and the semantic data store about 25 megabytes. Our experience has shown that for almost all systems, the combined sizes of these two data stores is less than ten times that of the source (and the ratio is typically smaller for larger systems and about half of this for non-object-oriented systems). This means that the extra space needed for the combined data store is quite reasonable and that it is practical to keep both in memory for fast access, as Desert does.

We note that these numbers also imply that our approach of keeping the data store in memory is practical even for large systems. Extrapolating from our experience, we estimate that a system with ten million lines of C source should have a data store size of about one gigabyte, well within the memory capabilities of modern workstations. Simple compressing techniques would easily cut the size of this database in half without any loss of performance. Larger systems could be accommodated by dividing the project into logical units, say representing the different executables or libraries.

The update time for the data store is minimal. When a single file is changed, the updates propagate into the store with no noticeable delay. When large numbers of files change, the data store typically processes files at the rate of one to three per second. This means that even complex changes (which typically occur only if everything must be recompiled or if the data store hasn't been updated for a long time) run fairly fast. (The update time is a small fraction of the compilation time, for example.) Even rebuilding the data store for the Desert system from scratch takes approximately ten minutes on a Sun workstation.

Keeping the data store in memory has also allowed our query response to be adequate for most purposes. The specialized queries used to find fragments or to do symbol table lookup run very fast (on the order of 10 milliseconds, including message overhead). Most sophisticated queries such as SQL requests from the visualization engine generally take time proportional to the size of the output, with the bulk of the time being spent on doing character output to generate the resultant file. The only queries we have had problems with are those that do not use the built-in indices for doing relational joins over very large (over 100,000 tuples) relations. As practical instances of these arise, we have either added indices or introduced new optimizations into the query engine.

There are difficulties with the data stores, however. The first is the need to compile all files so that the compiler generates cross-reference information. This does slow down compilation slightly and can use a significant amount of disk space in and of itself. A second issue is that the current data stores are not multithreaded and multiple query requests or update requests block other users. The consequent delays have not been a problem so far, but could be so in a multiuser environment with shared data stores and lots of activity.

4. Virtual Files

One of the principal features of data integration we wanted Desert to provide is the ability to offer the developer multiple views of the underlying system. Here we wanted to deliver a number of capabilities, including:

- *Arbitrary views:* Multiple views of a software system are useful for program understanding and during program maintenance. However, these tasks require a diverse set of views that cannot be determined in advance. Here the facility should make it easy for the user to define the specific views needed for these tasks using a general-purpose mechanism.
- *Diverse criteria:* In defining diverse views of a system, the user should be free to use any information available. This includes the various Desert data stores and the source files, as well as temporal information such as the set of lines with errors in the most recent compilation.

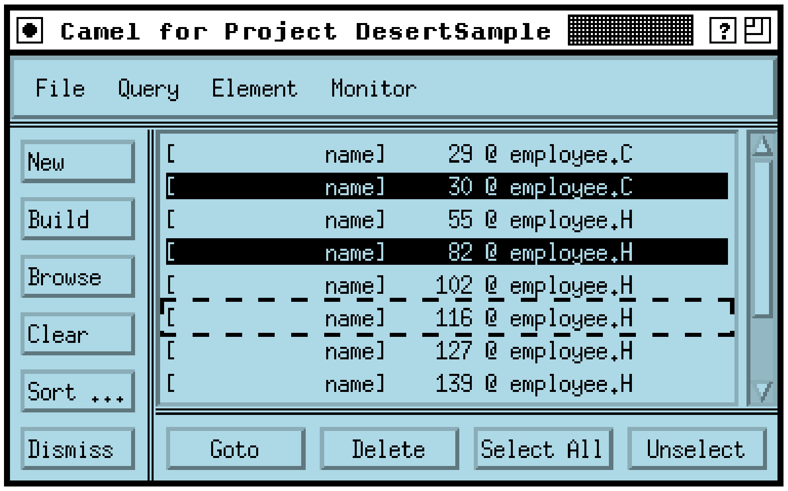
- *Arbitrary artifacts:* The views that can be constructed should be able to span the whole software engineering process. The user should be able to include specifications, design documentation and diagrams, code, user documentation, and error reports all in one view.
- *Editable views:* The views should be editable. While some applications of multiple views, for example, software understanding, generally require only read-only views, other applications, such as maintenance, are best handled with views that can be edited. However, supporting editable views is complex. It requires the editor to be aware of the context of each part of the view so it can be edited appropriately. It requires the ability to map changes in the view back to changes in the source objects. It also requires that views be managed correctly so they can be shared among multiple users.

4.1 Managing Virtual Files

The solution used in Desert is based on *virtual files*. These are temporary files that Desert constructs from a user-selected set of fragments. Virtual files can be edited and used as any other file. When a virtual file is saved, any modified fragments in that file are replaced in their original locations. This approach offers the above capabilities while maintaining an open environment based on files.

Fragments are a natural unit for defining system views. They provide a logical context for presentation in a view; they can be easily extracted and replaced; they can be defined for a wide variety of software artifacts; and they can be identified readily using information in the various data stores. Each virtual file consists of a file header identifying the context in which the file was created, individual headers for each fragment identifying the fragment, its location, and its lock status, and the fragments themselves.

In order for virtual files to be practical, the environment must support their creation and update. A Desert tool defines a virtual file by specifying the basic set of fragments for the file, either directly or indirectly as file-location pairs. Desert processes this list to ensure consistency and build the file. It first ensures that only fragments of appropriate types are added to a virtual file. The types allowed in a particular virtual file are specified when the file is first defined. If a fragment is requested of a type inappropriate for the given file, Desert substitutes the appropriate parent of that fragment. This lets the tool creating the virtual file specify the level of granularity to present to the user. Second, Desert ensures that only one copy of a particular fragment is included. This involves checking parents to ensure that a parent and its child are not both included. This is necessary to ensure the consistency of both the presented file and the result of editing it. If more than one copy of a fragment were present, the file could present multiple, different representations of the same source and it would be unclear, when the virtual file was stored, which representation should be used. Next Desert determines the initial lock status of each fragment, using both UNIX permissions and an underlying lock manager; it uses an optimistic approach to locking so that fragment locks are created only when needed, not when the virtual file is created. Finally, Desert builds the virtual file.



```
[           name]     29 @ employee.C
[           name]     30 @ employee.C
[           name]     55 @ employee.H
[           name]     82 @ employee.H
[           name]    102 @ employee.H
[           name]    116 @ employee.H
[           name]    127 @ employee.H
[           name]    139 @ employee.H
```

FIGURE 7. A view of a CAMEL search window.

When the virtual file is saved, Desert processes it, using the headers it generated to identify the fragments in the file. It then orders the fragments so that multiple fragments derived from the same file are considered from the back to the front to ensure that the fragment positions in the data store are valid for all the fragments in the file. Desert checks whether each fragment was actually changed by comparing the hash value of its new contents to the value stored in the data store. If the fragment was changed, Desert then replaces it in the original artifact.

4.2 CAMEL: Defining Virtual Files

For virtual files to be useful, the environment must provide the user with appropriate tools for manipulating them. Desert first attempts to make it easy for the user to define virtual file through a search and query tool, CAMEL, which enables the user to find locations across all artifacts that might be relevant to the task at hand. The front end, shown in Figure 7, displays a list of relevant locations, each of which consists of a reason, a line, and a filename. CAMEL lets the user construct this list by querying the various data stores or monitoring other tools. It also lets the user edit the list directly. The list can then be used to direct the editor to view a particular location or to construct a virtual file.

CAMEL lets arbitrary queries be defined through resource files. Each query displays a set of appropriate fields that the user can define. Any items defined by the user are used as selectors when the query is evaluated. Four such queries are currently defined, one for finding references to a name, one for finding definitions, one for finding call sites, and one for general text search. Queries can be used to add or remove items from the current list of tuples. We plan to extend the interface of this tool to let the user define new queries and query types interactively. The actual queries are sent and processed by the SEGO interface described in Section 3.1.

An additional mechanism provided by CAMEL involves monitoring events and translating them into tuples. The system looks for messages directed at the CAMEL tool. Such messages can either be generated directly by other tools or can be generated automatically by the message mapper when any arbitrary message is seen. We currently monitor for compiler errors and warnings as well as user selections in the various tools.

4.3 Evaluation

CAMEL and virtual files represent a simple approach to defining editable views of programs. Program views have long been considered appropriate for a programming environment. Some of these, such as Interlisp's Masterscope package [55] and Linton's OMEGA system [27], simply provide views of information stored in a program database. Garlan's work on views for tools [16] allowed the definition of arbitrary editable views, but did so from a tools viewpoint rather than from the users: the idea was to let each tool define its own interface to the program data rather than to give the user an editable program view. Several older environments such as Magpie [11], provided editable views of a single routine. More recent environments such as Poem [26] or IBM's Visual Age C++ provide similar facilities. Reps' recent systems [44] use slicing to construct an editable slice from a simple program, so that the user can edit the slice and then merge the result back into the code.

Our work attempts to provide a practical front end so that programmers actually work in terms of virtual files, an approach that has had limited success. The overall concept of virtual files is different from what programmers are used to. Desert does not enforce (or even encourage) their use. We have found that most programmers tend to work in terms more familiar to them and not to learn the new tools and concepts needed to use virtual files effectively. A concurrent issue is how much the programmer is willing to trust the system to construct the "proper" virtual file. A virtual file is most useful if it contains all the information desired and little else. This means that the data store used to create the file must be accurate and the context of the presentation must be appropriate. This becomes a problem because the data stores typically only have incomplete information on files currently being worked on.

Another issue is whether the context provided by a fragment is the right one. The overall routine or the class or structure definition is probably an appropriate context as long as these contexts do not get too long. This is typically a matter of programming style. It would probably have helped if the editor had been able to show programs at different levels of detail and show automatically those parts containing the lines targeted by CAMEL at the lowest level of detail and others at a higher level. Even simply highlighting those lines in some way would have made a better presentation.

A more general question is whether virtual files are a viable presentation model. Virtual files gather the relevant information into a single file. However, FrameMaker and other editors present this file so that the user can view only one part of the file at a time. The very nature of a virtual file means that the programmer is likely to want to view multiple portions, for example a definition and its use, simultaneously. A more flexible editor presentation, one that offered a table of contents and multiple views, would probably be preferable here.

Another general issue is the appropriateness of CAMEL as a front end for defining virtual files. It seems that a general-purpose front end is useful here. However, it also seems, from our limited experience, that programmers typically construct virtual files of a small number of types using limited information, and an interface that lets such files be created more directly might be more appropriate. We will evaluate this more thoroughly as we get more experience with the system.

Desert was designed to use virtual files primarily as temporary artifacts that are edited once and then disappear. An interesting alternative would be to create permanent virtual files offering the user different views of the software system. This raises a number of concerns, however, regarding consistency and how and when such files should be updated.

Perhaps a better approach to this problem would have been to view virtual files as dynamic, editable views of the underlying data store. Database views are typically defined as queries into the database. As such, although defined once, they can vary as the database changes and can be updated dynamically. Desert's virtual files are defined as a specific set of elements and hence cannot be updated dynamically if additional fragments become candidates for the virtual file. Database views also allow arbitrary manipulation of the data, while Desert restricts virtual files to contain exact copies of pieces of the files. The advantage of the Desert approach is it avoids the database view-update problem in which updates to the view are ambiguous with respect to the database. It also avoids most of the problems cited in Meyers' work on supporting different semantic views [29].

5. Common Editor

While fragments and the semantic data stores provide an internal basis for a more powerful environment, we felt that providing an external basis in the form of a common editor was equally important. We wanted Desert to offer the programmer a single context that could handle a wide variety of different software engineering tasks. At the same time we wanted the editor to take advantage of the semantic data stores and high-quality graphics displays.

From these objectives, we developed a set of requirements for program editing within the Desert framework. These included:

- *System-based editing*: The user should think of the editor as a front end to all the artifacts making up the system rather than to a single file. The editor should be aware of definitions that span files and searches should cover all relevant files.
- *Semantic-based editing.* The editor should make use of semantic as well as syntactic information and should help the programmer as much as possible. This means detecting both syntax and potential semantic errors as early as possible and providing feedback on such errors in a non-obtrusive way during editing.
- *High-quality formatting.* While most programmers today use machines capable of providing high-quality graphics, most program editors make little use of these capabilities.

- *A single common editor*. Programmers currently are forced to use multiple editors, one for editing programs, one for editing documentation, one for editing specifications, etc. An integrated programming environment would use a single editor for all documents and all phases of development. Such an editor has to handle diagrams as well as text, documentation as well as code. Where specialized editors are needed, for example a graphical editor for creating a user interface, these should be integrated with the common editor.
- *Integration with the environment*. The editor needs to be integrated into the overall environment so that it can talk to other tools and so that other tools can communicate through it.
- *Simplicity and familiarity*. Finally, we wanted an editor that programmers would actually use. This meant that the base editor must be familiar to them, preferably one they were using already. More importantly, we needed to be able to implement all the above facilities with minimal effort.

Our solution to these requirements was to use Adobe FrameMaker as the basis for the common editor. FrameMaker provides many of the baseline features we needed: it displays both pictures and text; it can display high-quality program views; and many programmers are familiar with it, using it for documentation or specifications. In addition, FrameMaker offers an application programming interface (API) that is relatively easy to use [9]. Through the API, one can integrate the editor with the rest of the system, provide additional functionality such as semantic-based editing, and provide live-links to other graphical editors.

While Desert uses FrameMaker, our intention was to demonstrate that the desired features could be provided to the programmer and not that FrameMaker was the ideal platform for doing so. Most of the extensions that we provide for FrameMaker could have been done for any other word processor with a suitable API and capabilities. (Microsoft Word, for example, could be used on a Windows platform.) As much as possible, the various extensions have been coded so that the FrameMaker-specific aspects are kept independent from the remainder of the code.

The common editor in Desert serves both as an integration tool and as a front end for new facilities. We have implemented a variety of APIs that extend the basic editor as shown in Figure 8. The rounded rectangles in the figure are the various APIs that talk to FrameMaker, the rectangles represent separate tools. Other than the message server (MSG) that acts as a communications engine, the separate tools are aligned with the API they are designed to support. The APIs currently available include:

- *FADE*: an annotation facility that extends the use of annotations demonstrated in the FIELD environment [39]. Annotations are a convenient and consistent way for tools to display information such as breakpoints in the editor and, at the same time, provide a standard interface whereby the user can make requests of other tools such as creating a breakpoint.
- *FAIL*: a general-purpose link package that lets the programmer install HTML and other forms of hyperlinks into a FrameMaker document. This provides additional functionality for maintaining documentation and other similar artifacts.

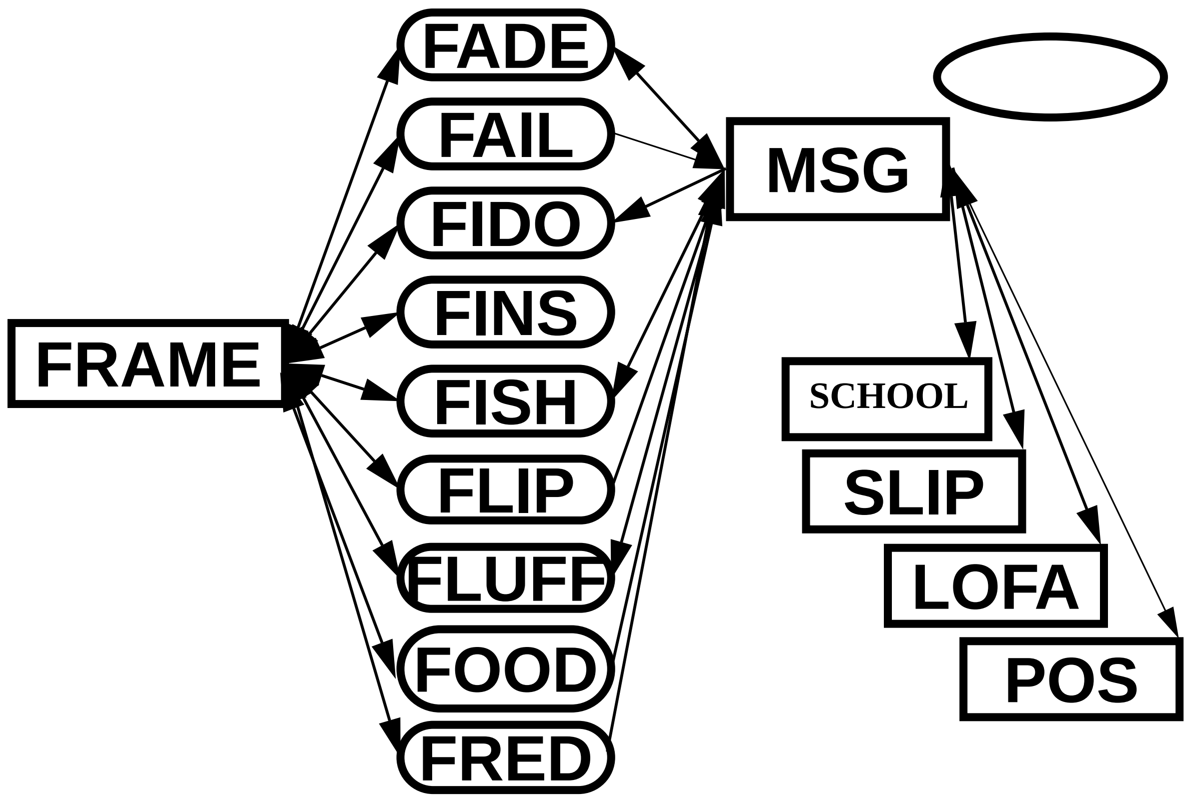


FIGURE 8. The architecture of the Desert program editing tools.

- *FIDO*: the basic interface between the message server and FrameMaker. It supports starting FrameMaker when needed by other tools in the environment and sending message-based requests to the various FrameMaker interfaces.
- *FINS*: supports live-links to embed other graphical editors in FrameMaker. It currently supports several varieties of object design and user interface editors.
- *FISH*: a first attempt at providing a shared editor using FrameMaker. It supports group whiteboard capabilities as well as common editing using a server.
- *FLIP*: supports Knuth-style web-based literate programming [23] using a server.
- *FLUFF*: provides fragment locking and virtual file support using a lock manager.
- *FOOD*: a first attempt at an object-oriented analysis facility using FrameMaker. A server is used to find nouns and verbs in a document.
- *FRED*: the primary interface for program editing. It provides all the necessary formatting capabilities, does incremental parsing and analysis, and provides semantic feedback as the user edits.

Some of these, notably FADE, FAIL, FIDO, and FISH, are written only as demonstration vehicles and are not complete implementations. The others are complete prototypes and are described in more detail in the following sections.

5.1 FRED: The Program Editing Interface

The first three requirements, system-based editing, semantic-based editing, and high-quality formatting, mandated that we provide extended facilities for program editing within the common editor. Here, in addition to providing well-formatted text, we offer the programmer non-obtrusive syntactic and semantic feedback. Syntactic feedback is provided through the use of indentation, with unexpected indentations being used to indicate syntactic problems, and through appropriate highlighting of keywords and contexts. Semantic feedback is provided through the use of formatted text, with different types of identifiers being displayed in different colors and with undefined symbols and other errors being displayed in red. In addition, the front end provides the programmer with implicit links between declarations and their uses throughout the system.

To handle dynamic text formatting as well as the appropriate syntactic and semantic feedback, we needed to parse the code as it was entered. While parsing on a keystroke basis is not new — it was done in the early 1980s in the COPE system at Cornell [1] — doing it outside of a syntax-directed editor, without direct control over the user's input, in the context of a powerful word-processing system, and in a language-independent manner is new.

A basic issue in designing such a parser is determining how the parse is to be represented internally. The parser needs to do incremental parsing to minimize the amount of work to be done on each keystroke. It needs to handle incorrect programs and also a variety of programming languages, including some (such as C++) that are notoriously difficult to parse. The obvious choice of representations, parse trees, has several drawbacks. They are much more complex than necessary, since we need only enough information for formatting. They are suitable when the program is correct, but are inadequate for incorrect programs. They are typically used to represent the language portion of the program and ignore comments and white space. Finally, parse trees imply that full parsing must be done, and even full incremental parsing can be complex and time-consuming. For example, adding a left brace in a C program would normally invalidate the remainder of the file and cause it to be reparsed.

FRED uses a simpler representation that meets our needs without the difficulties of parse trees. It represents the parse as two structures, a stream of tokens and a symbol table. The tokens serve as input for the simple parsing needed to maintain the symbol table, as the basis for finding the proper indentation for a line, and as the basis for formatting using.

In the next three sections we look in more detail at symbol table management, parsing, and formatting.

5.1.1 Symbol Table Management

Symbol table management in the editor interface enables the programmer to define and look up names inside scopes. It differs from traditional compiler-symbol table management in three respects. First, it provides incremental facilities whereby symbols can be dynamically defined and undefined to support incremental parsing.

This is done by including an *undefine* operation in the scope table and keeping track of all current implicit and explicit definitions.

The second difference is that FRED supports incomplete programs by maintaining the implicit type of undefined symbols. This is done by adding an *assume* operation that takes a name and the symbol type. If the name is already defined, the assume operation is ignored. If the name is undefined, this operation creates a definition in the outermost scope regardless of the scope it is called for, and sets a flag in that definition indicating that the name is assumed. If the name was previously assumed, then the symbol type of the previous definition and of this definition are checked to determine a new symbol type for a name. This is needed to handle names that can be used ambiguously, for instance type names that can be used as functions (for casting or constructors) and functions that can be used as pointer-to-function variables.

The third unique feature of FRED's symbol table mechanism is its connection with the system data store to facilitate lookup and cross referencing of names defined and used in other parts of the system. This is managed through the outermost scope. This scope automatically establishes a connection with the proper semantic data store, it determines the set of include files referenced by the file being edited and creates a lookup context in that data store consisting of these files and the files they include. When a name is looked up in the global scope, this data store is requested to find this name in this lookup context and the information returned is used to define the symbol.

The global scope is also used to manage dynamic cross-reference links based on uses and declarations. When the editor interface needs either the set of declarations or the set of references for a name that is defined or could be accessed externally, it sends a corresponding request to the data store.

The editor interface also provides a direct user interface to the symbol table through the symbol lookup dialog, which updates automatically as the user types, showing possible continuations to the current identifier and handling symbol completion. It also supports qualified lookup on demand. This is another example of aiding the programmer by providing access to appropriate semantic information.

5.1.2 Parsing

FRED parses the user's program for two reasons. The first is to maintain the symbol table and to locally link references to their declarations to support implicit links within the file. The second is to let the editor format the text to make it more readable.

Parsing is done in three phases. The first phase is scanning the source to produce a sequence of tokens. Figure 9a shows a simple source example that is tokenized as shown in Figure 9b. Scanning is done incrementally one line at a time, being extended to additional lines only when necessary.

Once tokens have been computed for a line, the line is parsed. The parser first goes through the tokens on the line to identify declarations and difficult-to-parse constructs. It saves its result by changing the token types returned by the scanner to

```
int fct(Bool x) {
   int a = 5;
   return x+a;
}
```

a) Sample Program

```
KEY_INT ID=fct LPR ID=Bool ID=x RPR
     LBR
KEY_INT ID=a EQ INT SEMI
KEY_RETURN ID=x OP ID=a SEMI
RBR
```

b) Original Token Stream

```
KEY_INT FCT_DEF=fct LPR_ARG1
     ID=Bool VAR_DEF=x RPR_ARG1 LBR
KEY_INT VAR_DEF=a EQ INT SEMI
KEY_RETURN ID=x OP ID=a
RBR
```

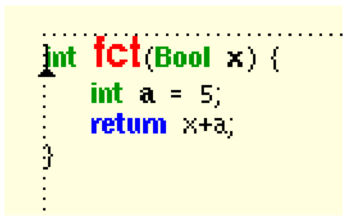
c) After Declaration Parsing

```
KEY_INT FCT_DEF=fct LPR_ARG1
     TYPE_REF=Bool VAR_DEF=x
     RPR_ARG1 LBR
KEY_INT VAR_DEF=a EQ INT SEMI
KEY_RETURN VAR_REF=x OP VAR_REF=a
RBR
```

d) Final Token Stream

```
Outer Scope
      Bool : Type (Assumed)
   File Scope
         fct : Function
      Function Scope
         Argument List Scope
               x : Variable
            Local Scope
                  a : Variable
```

e) Resulting Symbol Table



```
int fct(Bool x) {
   int a = 5;
   return x+a;
}
```

f) Resulting Display

FIGURE 9. An example of parsing inside the editor.

indicate definitions or special symbols. The result of this scan on the previous example is shown in Figure 9c.

The final parsing phase maintains the symbol table by doing a left-to-right scan through the tokens. Any token that starts or ends a scope causes the current scope to be updated. The name corresponding to a token identified as a definition in the previous pass is entered into the symbol table in the current scope. Finally, any identifier token not modified by the first pass is looked up in the current scope and context and its token type is changed to identify the actual symbol type. Symbols are first looked up in the current file. If they are not found, the system data store and other active editors for the project are queried to see if the symbol is defined at the outermost level or in an appropriate class. This gives the user the most up-to-date information and can be used to provide multiple-user support if the project is set up for multiple users (see Section 6.). Figure 9d shows the result of this second scan on the previous examples.

This approach to simplified parsing was implemented with C and C++ in mind but was designed to be language-independent. The implementation already isolates those parts that are language-dependent from the large body of common support code. We extended the original parser to handle Java in half a day's work. Other languages might be more difficult, but our intuition and preliminary analysis indicate that most languages will be easier to parse than C++. For the purposes of the editor, the parser need only maintain the symbol table and identify the symbol type of each identifier token. For most procedural and object-oriented languages, this involves finding declarations, building the symbol table, and then doing symbol table lookup, just as we have done for C++. Most languages other than C and C++ make it easy to identify declarations. The symbol table processing already provided by FRED was designed to handle most current and proposed symbol mechanisms. Extending it to other languages should not be a problem. We estimate that support for even a complex language such as Ada could be added in well under a week.

5.1.3 Formatting

Once parsing is complete, the text can be formatted. This is done in two stages. The first involves assigning a paragraph format to the line based on the tokens of that line and the second involves assigning a character format to each token based solely on the type of token. This simple approach is made possible by the parsing strategy, which encodes the result of the parse in the token types.

This approach is quite flexible. Our objective was to approximate the work of Baecker and Marcus [2] in designing a color display of the program text. Some changes had to be made in their recommendations to fit into the structure provided by word processors and the notion of the user directly editing the text. Accommodations also had to be made so that existing source code could be read in and so that a readable ASCII file could be regenerated.

Line-based formatting is used primarily to identify block comments so as to highlight them, as suggested by Baecker and Marcus. Blank lines served to identify valid breaking points so that functions can be kept on the same page as much as possible.

The key part of formatting is handled by assigning each token an appropriate character format. Different character formats are provided for different types of lexical units, as shown in Figure 10. In addition, different formats are used to describe different types of identifiers. Each of the identifier types shown in Figure 11 has three possible formats, one to indicate a reference to an identifier of that type, one to indicate a reference to an external identifier of that type, and one to indicate a definition of an identifier of the given type. Function declarations without a body (and similar method declarations) are formatted as external references rather than as definitions. The result of this can be seen in Figure 12. Type names, both built-in and user-defined, are shown in dark green. The names of functions being defined are highlighted in dark magenta in a large font to stand out from the rest of the text. Functions being called are shown in bold italics. Names being declared, both in the

Character Format	Description	Sample
Comment	Inline comment	*// comment*
Token	Single- or multiple-character token	++
SymToken	Single- or multiple-character token in Symbol font	*=
Keyword_Decl	Keyword as part of a declaration	**static**
Keyword_Stmt	Keyword starting a statement	**while**
Keyword_Type	Keyword representing a type name	**float**
Keyword	Any other language keyword	**this**
String	String or character constant	*"I'm a string"*
Constant	Numeric constant	1.2345
Id	Identifier of unknown type	identifier
Id_undefined	Undefined identifier	undef_id

FIGURE 10. The basic character formats used by the editor interface.

Identifier Type	Description	Local Reference	Global Reference	Sample Definition
Constant	Enumeration or named constant	SAMPLE	SAMPLE	**SAMPLE**
Function	Non-member function	***strcpy***	***strcpy***	**strcpy**
Field	Class, structure, or union field	manages	manages	**manages**
Label	Label as a goto target	label	label	**label**
Macro	Preprocessor macro	MACRO	MACRO	MACRO
Method	Class member function	***method***	***method***	**method**
Type	Class, structure, union or enum tag; typedef name	EMPLOYEE	**EMPLOYEE**	**EMPLOYEE**
Variable	Variable name	variable	variable	**variable**

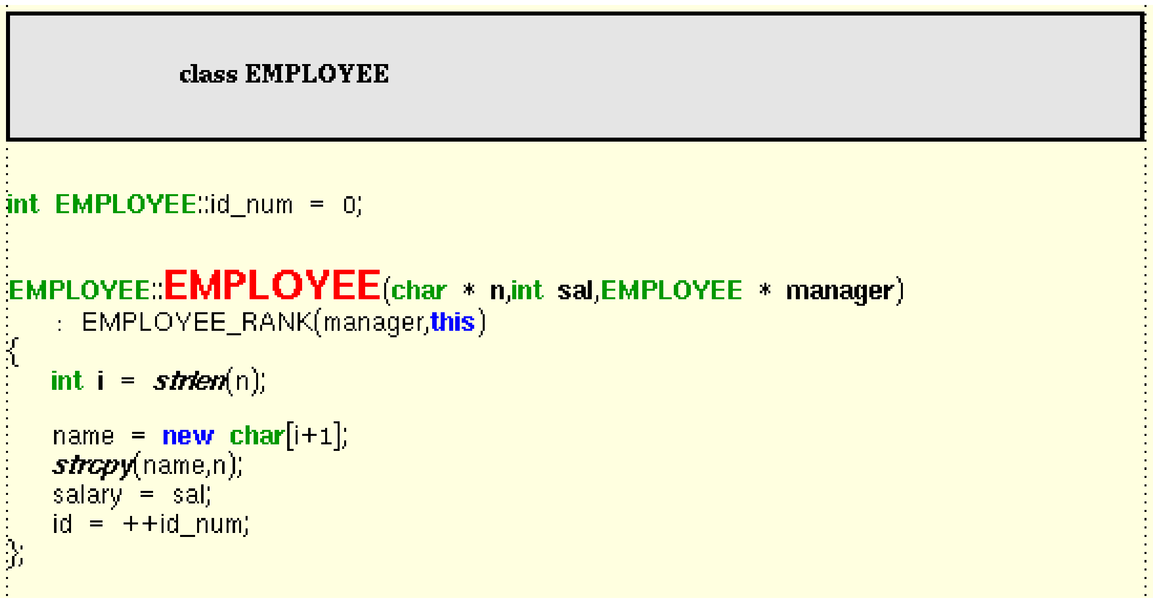
FIGURE 11. Formatting styles for different types of identifiers. Most identifiers have different formats depending on whether they are local references, global references, or definitions.

argument list and in declarations, are shown in boldface. Keywords are displayed in blue. All other identifiers are shown in a standard font.

5.2 FLUFF: Support for Virtual Files

One of the goals for our common editor was that it should be integrated into the environment. Since one of the goals of Desert itself is to support virtual files, we needed to ensure that the editor provide full support for such files.

Most of the virtual file support is embedded in the program editor interface described in the previous section. The header lines for a virtual file are parsed and formatted separately by this front end and are protected from editing. The program editor also sets up appropriate editing contexts for each individual fragments, so that name lookup and the like can be done appropriately within that fragment.



```
class EMPLOYEE

int EMPLOYEE::id_num = 0;

EMPLOYEE::EMPLOYEE(char * n,int sal,EMPLOYEE * manager)
   : EMPLOYEE_RANK(manager,this)
{
   int i = strlen(n);

   name = new char[i+1];
   strcpy(name,n);
   salary = sal;
   id = ++id_num;
};
```

FIGURE 12. Example of formatting in the editor interface.

Full virtual file support, however, also requires support for fragment-based locking. With whole files it is fairly easy for the editor or user to ascertain if the file is already being edited or in use. However, with fragments it is easy for even a single user unwittingly to create multiple views of a single fragment and change each one separately. Fragments also hold the promise of a finer granularity for sharing source code during cooperative development by letting multiple programmers edit different pieces of the same file safely.

Desert supports a simple form of locking based on fragments using the FLUFF editor interface and the LOFA lock manager. FLUFF uses optimistic locking to keep track of the lock state of each fragment in the current file. If the user attempts to edit a fragment that has not yet been locked, FLUFF attempts to lock that fragment by sending appropriate messages to the lock manager. If the fragment is locked by another file or another user, then FLUFF disables edits on the corresponding portion of the file. Otherwise, FLUFF updates the lock state and allows editing.

The lock manager, LOFA, implements a simple nested locking strategy with both read and write locks. It is modularized so that if in the future a more sophisticated locking strategy becomes necessary it could be replaced with a system such as PERN [4,21]. LOFA communicates with FLUFF and any other tools that require fragment locking through the global message server. This ensures that locks are maintained system-wide. The infrequency of lock requests ensures that this is not a performance bottleneck.

5.3 FINS: Insets for Software Artifacts

Another goal for our common editor was to provide common access to a variety of software artifacts. Several of these artifacts, particularly the graphical ones, are built and maintained using special purpose editors: OMT diagrams describing an object-oriented design are maintained with an object design tool; user interface designs are edited using an interface builder. Ideally such specialized editing should be included as part of the common editor. This is typically done today using a facility such as ActiveX that allows embedding a document of one type within another and uses a common front end for the different editors. FrameMaker, in its current state, does not support such a facility. Such facilities assume, however, that both the base editor and the specialized editor be designed with a common interface and facilities in mind. This is generally not the case in practice.

The FINS editor interface does the best it can in integrating a suite of diverse tools, using their interfaces and the inset mechanism that FrameMaker does provide. FINS manages two files for each inset interface. The first file is the input file to the tool, i.e. a file representing the software artifact. For tools that actually use a database such as Paradigm+ and Cadre's OMT tool, we generate a state file when the user saves the design and use this file to recreate the windows open at the time of the save. For tools that have a single data file, such as Builder Xcessory, Sun's VISU interface builder, and GE's *omtool*, we use the data file saved by the tool.

The second file managed by the front end is a file containing the image to display in the inset. This can either be a PostScript file or a X11 image file, and is generated in various ways depending on the tool's capabilities. The common inset interface detects when the tool input file is older than the image file and automatically generates a new image file. Integrating additional tools through the inset interface is simple and fast, taking less than half a day provided that the tool has a state or data file or one can be easily generated and that there is a logical way of generating an image file that can be incorporated into FrameMaker.

5.4 FLIP: Literate Programming Support

Literate programming [23] is an attempt to make programs readable by humans by integrating documentation and code in a single file. Since this coincides with one of the goals of the Desert environment, it is only natural that Desert provide support for literate programming.

Desert's support for literate programming is twofold. First, since FrameMaker can define both text and graphics and is commonly used for program documentation, Desert supports a straightforward form of literate programming by letting the programmer intersperse documentation and code in the same file using standard FrameMaker facilities.

Literate programming, however, is more than simply placing code and its documentation in a single file. Childs [8] provides a list of features that literate programming should support. These include generating problem descriptions and examining alternative solutions, breaking the program down into logical subdivisions such as chapters and sections, presenting the program in a logical order, and providing reading aids such as cross-references automatically.

To support literate programming along these lines, we implemented an additional interface, FLIP, that supports a traditional approach similar to that of other literate programming systems. In doing so, we make effective use of FrameMaker by letting the programmer use the formatting capabilities of the programming interface for code while still providing access to the full range of text and graphics for documentation.

Most literate programming systems utilize stylized commands in the code to indicate different sections of documentation and code. Our literate programming interface uses special FrameMaker paragraph types for this purpose. These paragraph types are all displayed with special visual attributes to indicate that they are not part of the document but rather define how the document should be interpreted.

In addition to this interface, our literate programming support depends on a separate server, SLIP, that does the underlying processing. When a file containing literate programming commands is saved, or when the user specifically requests an update of the literate programming portions from a file, FLIP sends information about each file to be generated, each container, each container extension, and the contents of the containers to SLIP, which in turn stores them in a directory that is specific to the current Desert project and is a database for literate programming information for the project. SLIP then makes it possible to regenerate or update all source-code files for a given project from this specialized database. An interface for triggering this update is provided in FrameMaker.

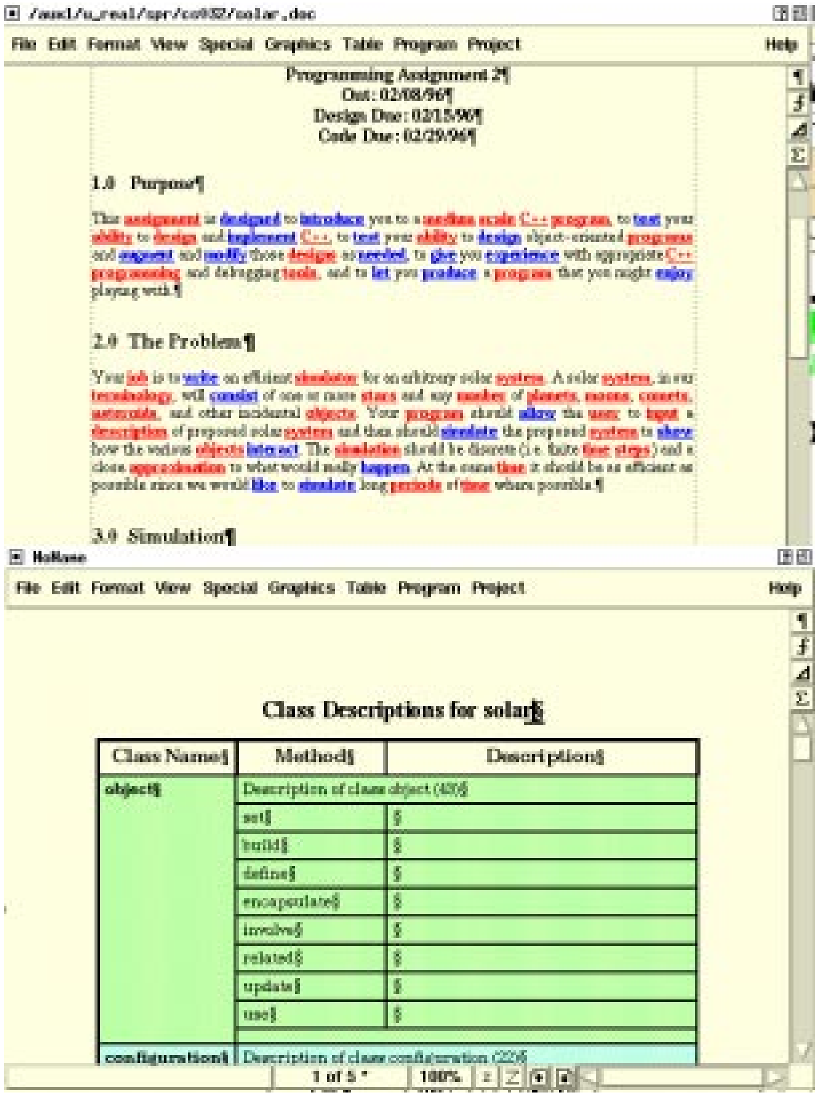
5.5 FOOD: Object-oriented Analysis and Design Support

To demonstrate the use of Desert in other phases of software engineering, we have included a small set of tools for object-oriented analysis and design. Some of this support involved integrating design tools such as OMT editors into the editor. To go beyond this, we added a tool for finding a candidate set of objects from a specification document and using this set to build an initial OMT diagram. Extending this tool with additional capabilities is one of our current research projects.

The first stage of object-oriented design involves determining the set of object classes that will be needed. Several techniques have been proposed to accomplish this, ranging from looking for nouns and verbs in the specification documents to constructing and analyzing scenarios. Our tool, the FrameMaker interface FOOD, uses the former approach since it can be easily automated.

FOOD uses an external package developed for natural-language processing to determine the parts of speech of each word in an arbitrary FrameMaker document [7]. A further analysis selects nouns that occur frequently as potential objects, taking into account synonyms, reference words such as “anything”, and word size. The verbs that act on these nouns then become the potential methods. FOOD builds a table of potential objects and their methods. The user can edit the table to refine the set of objects and methods or to add more information. The tool can then take the edited table and produce an initial OMT diagram using one of the OMT tools that the environment supports. This diagram can then be edited with the appropriate editor and included in the appropriate source or virtual files using the inset mechanism.

Two views of the use of FOOD in FrameMaker are shown in Figure 13. The upper view shows nouns and verbs in the document, nouns highlighted in red (the lighter underlined items in the figure) and verbs in blue (the darker underlined items). The


/aux1/u_real/spr/cs032/solar.doc
File Edit Format View Special Graphics Table Program Project Help
NoName
File Edit Format View Special Graphics Table Program Project Help
1 of 5 *
100%

Programming Assignment 2¶
Out: 02/08/96¶
Design Due: 02/15/96¶
Code Due: 02/29/96¶

1.0 Purpose¶

This assignment is designed to introduce you to a medium scale C++ program, to test your ability to design and implement C++, to test your ability to design object-oriented programs and augment and modify those designs as needed, to give you experience with appropriate C++ programming and debugging tools, and to let you produce a program that you might enjoy playing with.¶

2.0 The Problem¶

Your job is to write an efficient simulator for an arbitrary solar system. A solar system, in our terminology, will consist of one or more stars and any number of planets, moons, comets, asteroids, and other incidental objects. Your program should allow the user to input a description of proposed solar system and then should simulate the proposed system to show how the various objects interact. The simulation should be discrete (i.e. finite time steps) and a close approximation to what would really happen. At the same time it should be as efficient as possible since we would like to simulate long periods of time where possible.¶

3.0 Simulation¶

Class Descriptions for solar§

Class Name§	Method§	Description§
object§	Description of class object (40)§	
	set§	§
	build§	§
	define§	§
	encapsulate§	§
	involve§	§
	related§	§
	update§	§
	use§	§
configuration§	Description of class configuration (22)§	

FIGURE 13. FrameMaker windows showing the object-oriented design tool FOOD in action. The top window shows highlighting of nouns and verbs in a document. The bottom window shows a portion of the resultant object table.

bottom view shows part of the table of potential classes generated from this document, objects on the left and potential methods for these objects in the middle; space is left for more description of both objects and methods. The table is designed so that additional information such as relationships between objects can easily be added in the future.

5.6 Evaluation

Our experience to date has shown the power of using a single editing interface in a programming environment. The common editor makes editing easier for the programmer and provides a convenient front end to a wide variety of programming tools. The best feature we have found thus far is the use of formatting and unobtrusive semantic feedback (for instance, displaying undefined identifiers in red) during program editing. Once one gets used to these facilities, going back to a more "primitive" editor lacking them is bothersome. (The purely syntactic highlighting of Visual C++ or *emacs* noticeably lacks the accuracy and completeness that FRED offers.)

The use of FrameMaker as the common editor has been somewhat controversial. FrameMaker has all the facilities needed in a common editor. It lets us mix text and graphics for documentation, supports high-quality formatting of programs, and offers inset and link mechanisms for integrating the editor with other tools. These, combined with the fact that many programmers are already familiar with the editor because they use it for documentation or writing papers, made it a convenient choice. Moreover, its powerful API enabled us to implement a large number of tools.

On the other hand, FrameMaker was not designed for applications such as programming and the highly interactive extensions we have added. FrameMaker is large and relatively slow as editors go. The time to open a file is generally measured in seconds and the system is almost unusable on older hardware. We also had to take special measures to eliminate some of the bottlenecks that arose in implementing the FRED package. While we still have some performance problems in the current implementation, in terms of both speed and memory usage, we have gotten to the point where the editor is usable.

While FrameMaker provides Desert with many capabilities, it has restricted the interfaces we can offer. Insets in FrameMaker are somewhat primitive when compared to Microsoft's ActiveX and similar technology. FrameMaker's cross-reference links are also somewhat restricted, being difficult to activate and define. Comments in the program display had to use simple tables to achieve the desired background. This made the design and implementation of the program-editing front end more complicated than necessary. Annotations were also difficult to integrate into the document-oriented philosophy of FrameMaker. Finally, because FrameMaker essentially controlled the display, some of our user interface options are not optimal. For example, the dialog box for symbol completion would probably be better using a popup menu or cue, as in Visual C++.

Overall, however, our experiences with the use of a word-processing editor in general and FrameMaker in particular have been positive. We anticipate that other environments will move in this direction as these tools become more general and the machines they run on become more powerful.

6. Context Management in Desert

The Desert framework is designed to handle large-scale software engineering projects typically involving a number of programmers working on a large set of files to produce a smaller number of different binaries. Doing this requires the notion of a project.

The environment must recognize and utilize projects in several ways. First, the project must define the set of relevant source files that will be used in the data stores. Second, it must provide the information needed to understand how to scan and use these files, for example indicating where include files can be found or what compilation flags are to be used for particular files. Third, it must allow easy customization so that each individual programmer can have a private view of a complex project. Fourth, it must be readily adaptable, anticipating and tracking changes in the set of files and binaries without explicit commands from the user. Finally, it must let the programmers edit the project definitions as appropriate. Desert accomplishes these ends through the use of *contexts*.

6.1 Contexts

Desert defines a context as a set of files along with associated information. The files are identified with two sets of path names, one indicating paths to include in the context and one indicating paths to exclude. Each path name to include can either indicate an individual files or a directory. In the latter case all files in the directory and in any of its subdirectories are included or excluded as appropriate. This provides the adaptability cited above, with new files added to directories or files renamed within a directory being handled automatically. The set of path names to exclude are interpreted as regular expressions to give the user the most flexibility.

Focusing on the set of base files as is done in Desert rather than on target systems provides additional flexibility. In Desert, a collection of binaries can be included in a single context. For example, all the Desert tools (about forty binaries not including test programs) are included in the Desert context; similarly, the Forest context includes the underlying C++ support library used by Desert as well as a large collection of test programs. A context can also be built with no executable, as, for example, during the design stages of a system or when only the literate programming facilities of Desert are being used.

Desert contexts provide a general mechanism, associations, for providing information about individual files. Each association is a mapping from file name patterns to data. If a file name matches a given pattern, the associated data either replaces or is added to (depending on the type of association) the corresponding data for that file. For example, the INCLUDE association lets the user specify paths to be used to find include files while scanning and is additive, with multiple patterns adding separate include paths for a file. Other associations are provided for compilation options, for determining the appropriate source language, and to let contexts be integrated with a version control and a configuration management system.

Desert provides individualized contexts in two stages. It first supports two basic types of contexts: *global contexts* that are shared among a set of users and *local contexts* that are specific to a single user. Second, it lets a new context be defined relative to a base context. In this case the new context inherits the definitions of the base context and can provide additional definitions of its own as well as override the base definitions. This lets the overall information about a project be defined in a shared global context while letting individual programmers create a custom context with their own minor variations of the global information. It can also be used to create multiple global contexts representing different versions of a system.

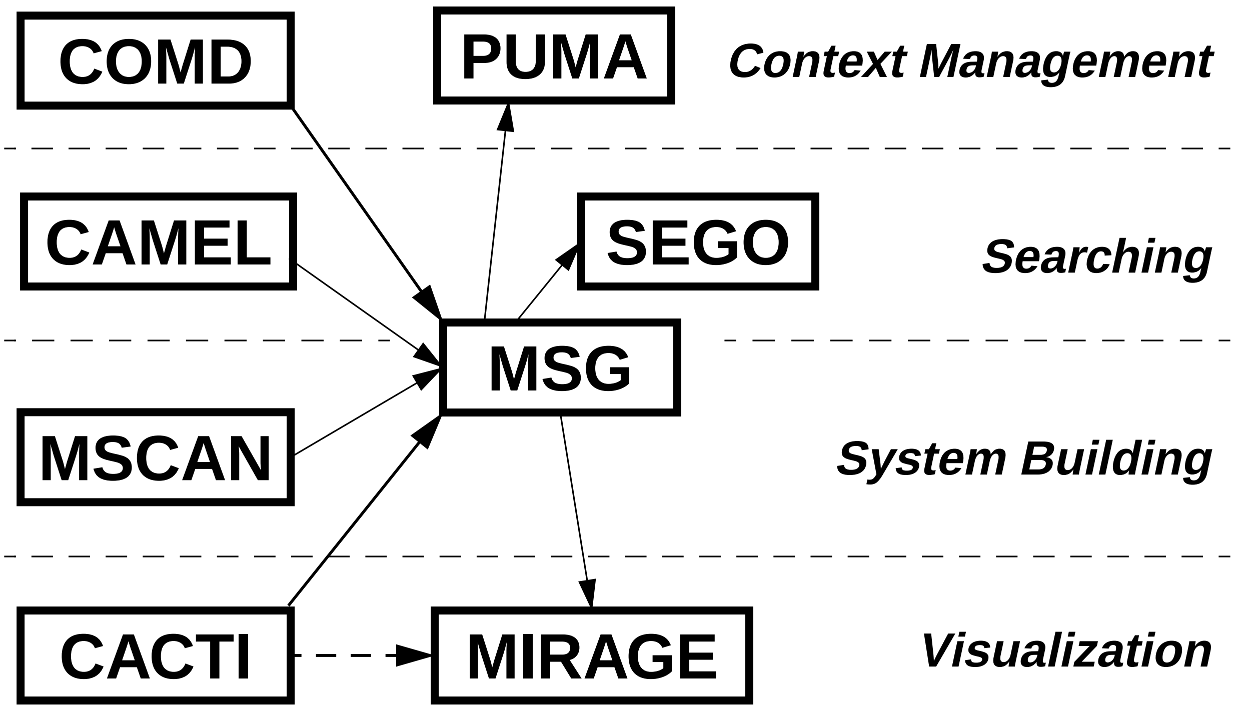


FIGURE 14. The tool architecture of the Desert environment. The message server connects the various tools to their servers and to the data stores and editing tools. COMD provides context-management services using the PUMA database server. CAMEL provides search and virtual file-building facilities using the SEGO query server as an interface to multiple data stores. MSCAN provides an interface to make, scanning the output to generate messages for compiler error and warning messages. Finally, CACTI and MIRAGE are new tools being developed for program visualization.

6.2 The Context Interfaces

Desert makes the information about contexts available both to the user and to other tools. The interface to other tools is provided by the PUMA front end as shown in Figure 14. PUMA runs as an independent process connected to the other tools using the message server. It lets other tools query and set information about the various contexts, and provides additional facilities for creating new contexts, finding the context associated with a given file, returning all files in a context, and determining if a given file is in a context.

The interface to the user is through the COMD tool. COMD was developed as a simple front end for context management, letting the user define, edit, and control updates to contexts. As shown in Figure 15, it provides a set of pulldown menus for this purpose as well as a display area showing what is in the context. For the display, we use the 3D facilities provided by the tools we developed for program visualization [43]. White nodes represent directories explicitly used in the context; black nodes represent directories explicitly excluded; otherwise a node's color represents the last time that node or any file in it was modified. The height of a node represents the number of files. The display can also show the particular files included in the database.

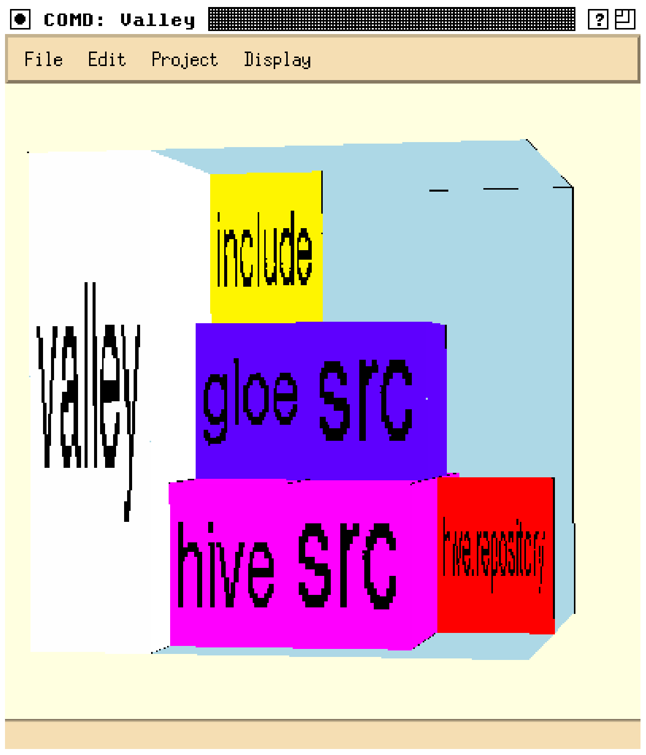


FIGURE 15. The COMD interface for a project showing its various directories. Colors code the date last modified of any file in the directory. Height above a midline shows the number of files while depth below the midline shows the total size of the files.

6.3 Evaluation

PUMA is actually a placeholder for what should be a complete configuration management system that can be used for both version management and system building. PUMA was designed to be general enough to serve as a back end for configuration management, but has never been used to its full capacity or with an appropriate set of front ends for system building or versioning. For example, it provides functionality for selecting which version of each file should be used using patterns similar to the Shape system [28].

The COMD interface was also designed for extension to a full front end for configuration management. Here we drew on previous experience with the *formview* tool provided by FIELD and the POEM system [26]. Many of the capabilities, for example specifying what include paths are to be used for different directories or files, are similar to what is needed in a configuration management package. Moreover, the display component of the tool was designed to support a variety of different displays, for example a file-dependency display or version tree display as well as the current directory view.

While both of these systems were designed to be extended, the fact that they have not been is one of the major drawbacks of the current Desert environment. Any configuration information needs to be specified twice, once for PUMA and once in a makefile. Most of this information should be specified at most once, since multiple specifications not only require additional work of the user but also can lead to inconsistencies.

The lack of a central system model with version and building information has also hurt other tools in the environment. For example, searches done in the editor should restrict themselves to files that are part of the same generated binary as the current file. One should be able to restrict queries in the CAMEL tool to a particular binary or subsystem or a particular version of the system. The editor should be aware of different versions of a source file and automatically find the proper one. Because of all this, we now feel that any new programming environment should be built with an integrated system model from the start.

The advantages of a central system model are also shown in the various process-centered environments such as those of the Arcadia consortium [53] and Marvel [4]. These environments provide a flexible framework for directing and augmenting the development process, and assume that the user provides a suitable model describing the system being developed. The existence of such a model lets these environments provide a proactive set of system management tools to assist the programmer. Once Desert has a full system model, adding such capabilities using either rules (like Marvel) or a process language (like Arcadia) would be a natural extension.

7. Related Work

Desert uses a combination of control, data, and presentation integration to approach a seamless environment. Control integration, a technique we pioneered for programming environments in the FIELD system [40-42], 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, Digital's FUSE, and Sun's Workshop. It is relatively inexpensive and allows the use of existing tools with only minor modifications. Moreover, as we demonstrated in FIELD, it can be completely compatible with the use of existing tools. However, it does not provide complete integration.

Data integration was first introduced in the early 1980s. It has been used to some extent in Ada programming support environments [32], in software engineering environments based on PCTE [6], in commercial tools such as ProCase [10], and in research environments such as Centaur [5].

Unfortunately complete data integration has proven difficult to achieve. An alternative has been providing a program database as another tool in the environment, either accessed directly or through control integration. Interlisp's Masterscope package used an internal database [55], and Linton proposed using relational databases [27]. More recently, the FIELD environment provides a cross-reference database of program information that is used by a variety of tools in the environment [25], and CIA and CIA++ represent environment-independent program databases for C and C++ respectively [20]. Sun's programming environment includes a similar tool, the source browser, that maintains its own database. Similarly, SGI's *cvstatic* 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 [48]. The use of an independent program database addresses some of the issues of data integration, but not all. Such databases 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 rarely handle multiple languages cleanly. Finally, most of these systems either require that the code is compiled in order to be included in the database (since the scanners are either built into the compiler or are effectively full language parsers), or only return approximate results, ignoring the scoping information needed to resolve duplicate names.

A more recent data-integration approach similar to that used in Desert is seen in the new version of IBM's Visual Age environment [33]. Here the compiler generates abstract syntax trees that are stored in a common repository and are available to the other tools. Unlike other attempts at data integration, the Visual Age repository is viewed as a cache, with the original program files viewed as the actual source of the data. This provides many of the benefits of data integration while preserving the openness of files. Desert uses a similar techniques, but with different goals in mind. Visual Age uses the database primarily to provide fast incremental compilation and is geared toward helping the individual programmer. Desert, on the other hand, emphasizes editing and tries to address a full range of software development issues. The two approaches are complementary.

The Desert emphasis on editing and presentation and its use of a common editor also builds on existing work. Many proposed and existing environments have been centered around the editor. This is natural since the editor is typically the tool used most widely by the programmer. FIELD, for example, integrates the editor into the environment using annotations to interact with other tools [39]. *Emacs* [19] represents the current UNIX approach; here the editor is extended to run other tools such as *make* to build systems and *dbx* to debug them, all within its own framework.

Desert extends such editor integration by letting a common editor support all phases of software development and by incorporating semantic information from other tools into the editing process. As such, it shares common elements with syntax-directed and language knowledgeable editors, literate programming, and hypertext editing. The closest previous environment is probably Cedar Mesa where a single specialized editor was used for both structured documentation and for programming [52,56].

Many editors have been written exclusively or primarily for programming. Many of these are syntax-directed editors, editors that parse the program and let (or force) the programmer work in terms of syntactic units of the underlying programming language. Syntax-directed editors were widely proposed and implemented in the early 1980s [11,13,14,38,54]. While some syntax-directed editors continue to be written, our general experience and that of others who have written and used them is that users do not generally edit in terms of syntactic constructs and the syntax-directed features often get in the user's way. Language-based editors [60,61] are a compromise that attempts to provide syntax-directed facilities along with standard text editing. Most current editors with syntax-directed features are of this type.

Programmer-knowledgeable editors such as the Programmer's Apprentice have also been proposed [59]. These attempt to use artificial intelligence techniques to provide direction to the programmer. More recently, the Pan system attempts to use sophisticated semantic knowledge to provide programmer feedback [3]. Neither of these approaches has been demonstrated as practical for realistically-sized programs.

More popular for programming are language-knowledgeable editors such as *emacs* [19] and those in Microsoft's Visual Studio. These editors know enough about the language being edited to do automatic indentation and simple error checking such as parenthesis balancing. Extensions included in such editors include a tags file for handling simple links between a definition and use (assuming the name is unique), and color highlighting based on simple patterns. Language-knowledgeable editors can offer many of the features that our editor interface does. However, they do not guarantee accuracy since the pattern matching (for indentation, links, and formatting) is approximate and only based on local information. Desert, even with the problems inherent to files currently being edited, does partial parsing and uses global information from the data stores to achieve significantly better accuracy. These editors are also not fully functional word processing systems that can display and combine graphics with text.

Other work related to ours includes work on literate programming [23,24,34,35], which is a general attempt to use a single file for both documentation and code. A preprocessor extracts the code from the file when compilation is needed. Other preprocessors can extract documentation, function headers, or other relevant information. Our editor interface is built on top of the commercial word processor FrameMaker, which in turn supports much of this directly. This is closer to the approach used in the Cedar Mesa environment where the Cedar editor used the document structure and document tags to distinguish code and comments in the same document [56]. As described in Section 5.4, we also provide extensions that support web-style literate programming directly.

Another area of related work involves the use of hypertext editors and browsers for programming. This has been advocated by several researchers as appropriate technology. It is becoming more common with the advent of HTTP and editors built for the World-Wide Web [15] that let the user create explicit links between the various parts of the program. Our approach can provide similar facilities, using the hypertext capabilities of FrameMaker for explicit links. Desert goes beyond this to create implicit links dynamically, for example letting the user click on an identifier that is not an explicit link and then computing where the definition or references to that identifier are using the internal parse and the external data store. Multimedia insets are supported both by HTTP and by FrameMaker. Hypertext-based tools such as *javadoc* preprocess a set of source files to produce a read-only linked view of the software. Many of the links that are created here are available automatically within Desert. Others items, such as automatically generated documentation, could be built within Desert as an additional tool.

Wasserman [58] and Thomas and Nejmeh [57] have both proposed extending the three basic types of integration addressed in Desert with the notion of process integration, measuring how the tools interact with a particular software development process. This generally involves ensuring tool compatibility and showing that the tools interact well with a given development process. Desert does not yet address this aspect of integration. However, incorporating additional tools to control and facilitate a software development process, a step integral to process integration, should be relatively easy to do within the Desert framework using the current control integration facilities and the data available in the various data stores.

Finally, the processing done in the Desert editor involves incremental parsing and semantic analysis. Incremental parsing typically shows how to extend standard parsing techniques to support incrementality [18]. A variety of techniques have been proposed for incremental semantic analysis, including attribute grammars [12], model-based functions [37], unification [47], and functions [22]. All these techniques attempt to preserve full semantic information and depend on having the full, error-free program available. Our simplified techniques provide the information needed for editing without the cost of maintaining the more detailed information needed by a compiler. Moreover, they work well in the presence of syntactic and semantic errors.

8. Experiences and Conclusions

Desert is currently implemented as a research prototype designed to get experience with and evaluate the potential of the various concepts it introduces. Our concerns to date have focused on whether such an environment can be made to work, and, if so, how to fit the various pieces together. We have been particularly interested in the potential uses of the various data stores and a common editor and in finding ways for the environment to improve programmer productivity.

At present Desert consists of about 250,000 lines (five megabytes) of C++ code evenly split between the actual implementation of Desert and general support facilities (templates, Motif interface, data store framework and query engine, Java parser). The actual implementation of the data store and of all the scanners each involve about 12,000 lines of code. The current search tool implementation involves 4,000 lines for CAMEL and 2,000 lines for SEGO. The FRED program-editing interface, the largest single component, is about 30,000 lines of code. The implementation has been done over four years.

The current environment has been used for its own development, for developing a number of related projects and course software, and, to a modest extent, in a small set of projects at Brown. This use, although quite limited, has demonstrated that the ideas are practical and scale to large software systems. Moreover, they have provided interesting feedback and ideas for possible future extensions.

The major strengths of the environment has established from the point of view of the environment developer are:

- A flexible control integration mechanism that makes it easy to add new tools;
- The introduction of fragment integration as a basis for finer-grain practical data integration among tools;

- Tool-accessible data stores with fast response and access to semantic information;
- Integrated support for creating and saving virtual files;
- A powerful word processor as a basis for presentation integration and for tool integration in general; and
- A context database supporting flexible project definition and file-base associations.

The strengths from a potential users point of view have been:

- Support for working in terms of virtual files including the ability to lock files at the fragment level for cooperative development;
- A common query interface that can do semantic searches throughout a project for either finding locations or for building virtual files;
- A single editor for most aspects of software development;
- Unobtrusive semantic feedback while editing; and
- An adaptive interface for context management.

The major weaknesses of the current environment and thus the primary directions for future work, as noted in the evaluation sections of this paper, are the lack of a system model for the data stores and the minimal use of virtual file-based presentations. We expect to pursue various ideas in both of these areas. For example, we are looking into different front ends that would allow a more structured and flexible presentation schema, possibly based on XML. We will also use the current environment as a basis for studying program visualization tools (using the information available in the various data stores) and intelligent program editing and presentation mechanisms.

We also plan to get significantly more experience with the Desert environment. We will begin to encourage its use in-house for student projects in a variety of software engineering courses. We are also making the system available, in both binary (for Solaris 2.X) and source form, at

```
http://www.cs.brown.edu/software/desert
```

As we get more in-house experience with the system, we will encourage and support outside users.

9. References

1. James Archer, Jr. and Richard Conway, “COPE: a cooperative programming environment,” Cornell TR81-459 (June 1981).

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

3. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, “The Pan language-based editing system for integrated development environments,” *ACM Software Engineering Notes* Vol. **15**(6) pp. 77-93 (December 1990).

4. Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman, “An architecture for multi-user software development environments,” *Software Engineering Notes* Vol. **17**(5) pp. 149-158 (December 1992).

5. 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).

6. 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).

7. Eugene Charniak, Curtis Hendrickson, Neil Jacobson, and Mike Perkowitz, “Equations for part-of-speech tagging,” pp. 784-789 in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press/MIT Press, Menlo Park (1993).

8. Bart Childs, “Literate programming, a practitioner’s view,” *TUGboat*, *Proceedings of the 1991 annual meeting* Vol. **12**(3) pp. 1001-1008 (1991).

9. Frame Technology Corporation, *Frame Developer’s Kit Programmer’s Guide*, Frame Technology Corporation (May 1995).

10. PROCASE Corporation, “SMARTsystem Technical Overview,” PROCASE Corporation (1989).

11. Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, “Viewing a programming environment as a single tool,” *SIGPLAN Notices* Vol. **19**(5) pp. 49-56 (May 1984).

12. Alan Demers, Thomas Reps, and Tim Teitelbaum, “Incremental evaluation for attribute grammars with application to syntax-directed editors,” *Proc. 8th ACM Symposium on Principles of Programming Languages*, pp. 105-116 (1981).

13. Veronique Donzeau-Gouge, Gerard Heut, Gilles Kahn, and Bernard Lang, “Programming environments based on structured editors: the MENTOR experience,” in *Interactive Programming Environments*, ed. E. Sandewall,McGraw-Hill, New York (1984).

14. Robert J. Ellison and Barbara J. Staudt, “The evolution of the GANDALF system,” *Journal of Systems and Software* Vol. **5**(2)(May 1985).

15. James C. Ferrans, David W. Hurst, Michael A. Sennet, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang, “HyperWeb: a framework for hypermedia-based environments,” *Software Engineering Notices* Vol. **17**(5) pp. 1-10 (December 1992).

16. David Garlan, “Views for tools in integrated environments,” pp. 314-343 in *Advanced Programming Environments*, Springer Verlag (1986).

17. David Garlan and Ehsan Ilias, “Low-cost, adaptable tool integration policies for integrated environments,” *Software Engineering Notes* Vol. **15**(6) pp. 1-10 (December 1990).

18. Carlo Ghezzi and Dino Mandrioli, “Augmenting parsers to support incrementality,” *JACM* Vol. **27**(3) pp. 564-579 (July 1980).

19. James Gosling, *Unix Emacs*, Carnegie-Mellon Computer Science Department (August 1982).

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

21. George T. Heineman and Gail E. Kaiser, “An architecture for integrating concurrency control into environment frameworks,” *17th Intl. Conf. on Software Engineering*, pp. 305-313 (April 1995).

22. Gail E. Kaiser, “Semantics for Structure Editing Environments,” Ph.D. Dissertation, Carnegie-Mellon University (1985).

23. Donald E. Knuth, “Literate programming,” *The Computer Journal* Vol. **27**(2) pp. 97-111 (1984).

24. Donald E. Knuth, “Literate Programming,” Stanford University Center for the Study of Languages and Information, Stanford University (1992).

25. 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).

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

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

28. Alex Mahler and Andreas Lampen, “An integrated toolset for engineering software configurations,” *SIGPLAN Notices* Vol. **24**(2)(February 1989).

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

30. Sun Microsystems, Inc., *The C++ Application Binary Interface*. 1995.

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

32. 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).

33. Lee R. Nackman, “An overview of Montana,” *IBM Research*, (1996).

34. Norman Ramsey, “Literate programming: weaving a language-independent WEB,” *CACM* Vol. **32**(9) pp. 1051-1055 (September 1989).

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

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

37. Steven P. Reiss, “An approach to incremental compilation,” *Proc. SIGPLAN ’84 Symposium on Compiler Construction*, (June 1984).

38. Steven P. Reiss, “PECAN: program development systems that support multiple views,” *IEEE Trans. Soft. Eng.* Vol. **SE-11** pp. 276-284 (March 1985).

39. Steven P. Reiss, “On the use of annotations for integrating the source in a program development environment,” in *Human Factors in Analysis and Design of Information Systems*, ed. R. Traunmuller,North-Holland (1990).

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

41. Steven P. Reiss, “Interacting with the FIELD environment,” *Software Practice and Experience* Vol. **20**(S1) pp. 89-115 (June 1990).

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

43. Steven P. Reiss, “An engine for the 3D visualization of program information,” *Journal of Visual Languages*, (December, 1995).

44. Thomas Reps, “Demonstration of a prototype tool for program integration,” U. Wisc.-Madison Computer Sci. Dept TR 819 (January 1989).

45. Ron Rivest, “The MD5 message-digest algorithm,” MIT Laboratory for Computer Science and RSD Data Security, Inc. (April 1992).

46. Dick Schefstrom and Ger van den Broek, *Tool Integration*: *Environments and Frameworks*, John Wiley and Sons (1993).

47. Gregor Snelting and Wolfgang Henhapl, “Unification in many-sorted algebras as a device for incremental semantic analysis,” *Proc. 13th ACM POPL*, pp. 229-235 (January 1986).

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

49. Robert Stockton and Nick Kramer, “The Sheets hypercode editor,” Carnegie Mellon University (1998).

50. Kevin Sullivan and David Notkin, “Reconciling environment integration and component independence,” *Software Engineering Notes* Vol. **15**(6) pp. 22-33 (December 1990).

51. SunSoft, *Tooltalk 1.1.1 User’s Guide*. November, 1993.

52. Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann, “The structure of Cedar,” *SIGPLAN Notices* Vol. **20**(7) pp. 230-244 (July 1985).

53. Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young, “Foundations for the Arcadia environment architecture,” *SIGPLAN Notices* Vol. **24**(2) pp. 1-13 (February 1989).

54. Tim Teitelbaum and Thomas Reps, “The Cornell program synthesizer: a syntax-directed programming environment,” *CACM* Vol. **24**(9) pp. 563-573 (September 1981).

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

56. Warren Teitelman, “A tour through Cedar,” *IEEE Software* Vol. **1**(2) pp. 44-73 (April 1984).

57. Ian Thomas and Brian Nejmeh, “Definitions of tool integration for envrionments,” *IEEE Software* Vol. **9**(2) pp. 29-35 (March 1992).

58. Anthony I. Wasserman, “Tool integration in software engineering environments,” pp. 137-149 in *Software Engineering Environments*: *Proc. Int’l Workshop on Environments,* ed. F. Long,Sprinter-Verlay (1990).

59. Richard C. Waters, “The programmer’s apprentice: knowledge-based program editing,” in *Interactive Programming Environments,* ed. D. R. Barstow, H. E. Shrobe and E. Sandewall,McGraw-Hill, New York (1984).

60. Jim Welsh, Brad Broom, and Derek Kiong, “A design rationale for a language-based editor,” *Software Practice and Experience* Vol. **21**(9) pp. 923-948 (September 1991).

61. Steven R. Wood, “Z - the 95% program editor,” *SIGPLAN Notices* Vol. **16**(6) pp. 1-7 (June 1981).