Fragments: A Mechanism for Low Cost Data Integration

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

1.0 Abstract

Control integration has become widely used as the primary means for combining a variety of programming tools in an environment. Data integration, on the other hand has not been nearly as successful. The primary reason for this is the high cost of data integration when it is viewed as a program database. Both the complexity of the needed database system and the high cost of modifying all the programming tools to use the database have made data integration prohibitively expensive. In this paper we propose a new means for data integration based on a database of fragments. Fragments are references to portions of a file. They have a source, type and associated attributes. The database and fragments are accessed through an object-oriented query language. This mechanism promises to provide most of the capabilities typically associated with data integration without the large costs. It allows the use of existing tools and multiple languages and can offer full support for the software engineering process.

2.0 Motivation

Message-based communication has been widely successful as an integration mechanism for programming environments. It allows a set of independent programming tools to effectively communicate with each other, providing the user with the impression of a seamless environment. However, because it only integrates tools that are running, has no notion of history or time, and has limited information about the system being worked on, it is difficult to provide some of the tools that are desirable in a programming environment and it is difficult to extend it to the multiuser, long-term domain of software engineering environments.

The alternative to message-based control integration has been data integration. Here the different tools in the environment share data through a common database. Data integration augments control integration by providing additional facilities that enable tools to know more about the system and to share data over time. The benefits of data integration to users can be summarized in the facilities that it provides. These include:

- Allowing the rapid selection and replacement of relevant portions of the program for editing and viewing based on semantic information, for example functions called from a given function. This enables the user to view the program as a hypertext document, moving easily from a use to a definition, from code to documentation, from error reports to code fixes, etc.
- Supporting the notion of links between portions of a system. This includes the implicit, semantic-based links described above as well as explicit, user-defined links such as where in the code a particular requirement is met.
- Allowing for traceability of links. It should be possible to create a link from a requirement or a specification or documentation to a section of code and have that linked automatically maintained through the life of the system.
- Implicitly understanding the dependencies of one part of the system on another. This allows intelligent system rebuilding by determining the effect of an editing change and can be used to forewarn the programmer about the effects of potential changes.
- Viewing the system as a collection of diverse documents not all of which are source code. Documentation, requirements, specifications, test cases, error reports, etc., are as much a part of a software system as is the source code.
- Allow the specification and recording of data associated with semantic program units by one tool for use in another. This can be used for information sharing as well as direct tool support. The compiler can store information about the program semantics that can then be used by other tools so they don't have to recompute it. Programmers might want to indicate that they want to be informed of any changes to a program unit. This information would also be used by a version management system as well as an intelligent editor.

In addition to functionality, other important ingredients of an integration mechanism for programming environments are openness and simplicity. Control integration has been as successful as it has because it encourages an open environment, supporting a wide range of systems, languages, and tools, and because it is simple to implement and use. For data integration to be successful as well, it must perform similarly. This means that it must:

- Support multiple languages. Existing programs are developed in multiple programming languages, either on purpose or because they are derived in part from previous systems or use libraries that are implemented in different languages. Moreover, portions of the system other than source code are each written and maintained in their own language.
- Use existing tools. Developing a new array of programming tools would be prohibitively expensive. Moreover, existing programs often use nonstandard tools such as lex, yacc, or other preprocessors.
- Be compatible with the existing environment. A new environment should allow a step-wise migration from the current environment. It should not force programmers to throw away all their previous tools and habits in order to use any of the new environment. Moreover, the environment should conform to the programmers. It should not dictate their tools, be it the editors or version management tools or languages, or anything else.
- Make it easy to add tools. This is needed to provide full support for existing tools, to integrate tools from a variety of vendors, and to provide enhanced capabilities through new tools.

Our research is aimed at achieving the benefits of data integration without the costs. In particular, we want to provide the facilities typically associated with data integration with an open and simple mechanism. Our approach is based on several insights:

- There is a basic program unit that is relatively large. While the traditional unit of a file is too large for tight integration, using the basic syntactic constructs of the language (such as Diana for Ada [7]), requires more information than is necessary and is too language specific. For most applications it is sufficient to break a program down into language-independent logical units such as functions, class definitions, declarations, etc. The other documents that compose the system can be broken down similarly. We call this basic unit a *fragment*.
- In order to support existing tools, the environment must preserve the existing frameworks. This means that the original files, be they source code, man pages, FrameMaker documents, etc., must be maintained as the primary representation.
- We need to have the ability to associate arbitrary data with fragments. Some of this data will be updated each time a fragment is updated. Other data will be updated whenever a tool is run, for example semantic information from the compiler. Other data will be more permanent, for example the original author of a fragment or a link from requirements to code. This data is the primary means for data-based communication among the tools.
- All links between fragments can be represented indirectly using data. By avoiding the need to store explicit links between fragments, we can greatly simplify the mechanism and can insure that updates can be handled quickly.
- There is some specialized semantic information that must be associated with fragments. In particular, the tools need to determine dependency information that can be represented in terms of definitions and uses. Rather than maintaining definition-use chains, however, it is sufficient to save references, definitions, and scope information and to compute the definition corresponding to a particular use as needed.

The basic idea behind our mechanism that we call *fragment integration* is to maintain a simple in-core database of information about fragments while leaving the body of each fragment reside in its original file. The database is automatically updated either by tools or by separate processes or scanners that run after the tools have finished. Tools can issue queries into the database to get the information that they need. Fragments can be quickly identified, extracted, and replaced in their files.

The next section of this paper describes the background for fragment integration, detailing the current alternatives of control and data integration, simple program databases, and wrapper-like interfaces. The following section details the basic concepts of fragment integration. Next we cover the problem of identifying fragments and the fragment database. We conclude with a description of how new tools will use the fragment facilities and our experiences to date with fragments.

3.0 Background

Integration in a programming environment today is usually viewed as a combination of control integration and data integration. Most commercial environment rely solely on control integration. PCTE [1] and evolving environments are emphasizing data integration as a needed facility.

Control integration, a technique that we pioneered for programming environments in the FIELD system [11], is very effective for integrating a concurrently running set of tools. It is the basis for most current commercial programming environments including Hewlett-Packard's Softbench and Sun's Tooltalk. It provides a backbone for tool communication through a central message facility and broadcast messaging. It is relatively inexpensive and allows the use of existing tools with only minor modifications or through simple wrappers. Moreover, as we demonstrated in FIELD, it can be completely compatible with the use of existing tools. While control integration can make an independent set of tools look like a single unified framework, it does not provide complete integration, and does not store information over time. It also forces all the tools to have a common reference framework, typically a file name and line number, which limits its applicability. Control integration also only provides surface integration among the tools — it does not provide for any permanent shared data or for integrating multiple users over time and space.

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

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

An alternative to complete data integration has been to provide a program database as another tool in the environment. Interlisp's Masterscope package used an internal database [14]. Linton proposed using relational databases [5]. More recently, the FIELD environment provides a cross-reference database of program information that is used by a variety of tools in the environment [4], CIA and CIA++ represent environment-independent program databases for C and C++ respectively [3]. Sun's programming environment includes a similar tool, the source browser, that maintains its own database. The use of an independent program database addresses some of the issues of data integration, but not all. They are limited to the source code and do not extend to other aspects of software development. They provide detailed information about variables, types, etc., but it is difficult for the systems to handle multiple languages cleanly. Finally, these systems require that the code compile in order to be included in the database since the scanners are either built into the compiler or are effectively full language parsers.

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

Another approach that is related to ours is literate programming [8,9]. Here a the user creates a single file that contains documentation and source code intermixed and various tools exist to extract the source for the compiler or the documentation for TEX. This approach provides the user with a very readable program, but does not easily scale up for large systems or handle other aspects of software engineering. One of the benefits of our approach is that we should be able to simulate much of literate programming by choosing documentation and code fragments from their existing files, presenting them to the user in a single, editable form, and then extracting the fragments from the result and storing them in their original files.

Our approach to integration is based on providing an interface to the original source files that emulates data integration. The general concept of providing a wrapper for a file or system or of providing a view of an object has been around for a long time. It can be seen in current technology in the interfaces that are being developed for Mosaic. Here there exist virtual files at various sites that when accessed as a file, actually invoke a computation of some sort. Other related work can also be seen in databases that try to abstract information from files such as the Rufus system [13].

4.0 Fragment Integration

Our approach to integration combines standard control integration with a much simplified form of data integration that we call *fragment integration*. Rather than storing everything about the system in a database, we identify fragments of the original file that are significant, and store references to these along with information about them. This information can be used to establish relationships between the fragments. Fragments can be identified differently for different languages, so that fragments can exist for programs as well as for textual documentation or for formal specification languages. The body of the fragments still resides in the original files and can be accessed by current tools without modification. In addition, our mechanism provides tools with access to individual fragments and with a query interface to the data associated with fragments.

Fragments are meant to describe significant portions of the source code or other material, especially those portions of a file that identify a scope. Fragments for a C++ program include files, function declarations, class definitions, and variable, member, and type declarations. Fragments for documentation written in FrameMaker include title, sections, and subsections. Fragments for a makefile include each make rule and macro definition. Fragments can be properly nested in other fragments.

Each fragment has a set of associated properties. It has an associated source, i.e., references a portion of some file in the system. It can be both named and typed. Each fragment is associated with a fragment scope and each fragment type defines a set of scope rules. Each fragment can specify names that are either defined or referenced in the fragment scope. For a programming language, the fragment scope corresponds to the associated program scope and the names correspond to entities being declared or used in the program scope. Finally, fragments have a set of attributes. These represent associations between a name and a value. They can be defined when the fragment is created or can be defined or modified by applications that access the fragment.

Relationships among fragments are defined in terms of these fragment properties using a simple query language. Examples of relationships would be physical containment, definition dependencies, documentation linkage, and call site to function body linkage.

Fragments can provide significant additional functionality to a programming environment. They can be used to simulate much of what data integration is proposed for with much less overhead and only slightly more computation. Fragments can be used to provide an intelligent configuration management tool where recompilation is dependent on definitions changing. Fragments can be used in an intelligent editor, one that provides ready access to referenced routines or definitions. Fragments can be used for hypertext-style editing of a combination of program source, documentation, requirements and specifications. Fragments can be used for program restructuring by composing new files from fragments of the old ones. Fragments can be used for program understanding and reverse engineering by identifying the basic constituents of a system and the relationships among them. Fragments can be used by a programmer's assistant tool to automatically create new fragments that anticipate future need, for example, creating a stub function automatically when the definition is created.

The logical extension of these concepts is using fragments to facilitate a new approach to programming. Rather than viewing a program as a linear document restricted to files in the file system, fragments would allow viewing the program as a dynamic, electronic document. For each particular problem a programmer faces, whether it involves writing code for a particular function, defining a class, debugging, or tracing requirements through the lifecycle, the environment could construct a custom document that contains those portions of the system that are relevant, and present this to the programmer in the most effective manner. The programmer could then work directly with this specialized document.

Fragment integration is a low-overhead mechanism because it deals with logical program units rather than low-level syntax and because fragments are only referenced indirectly. The size of the fragment database will be relatively small compared to the database needed for full data integration (it is about the same as the size of the combined source files). The high-level notion of fragments makes it relatively straightforward to write scanners for various languages that identify the fragments a file contains. This makes extending fragment integration to multiple-language systems and to other phases of software development simpler. Computing relationships among the fragments dynamically using fragment properties allows a highly modular fragment database that is easy to update. When a file is changed, we need only update those fragments that were in the previous version of the file and those that are identified in the new version. Most importantly, fragment integration will be compatible with the use of existing tools. Because the original files still exist and are still used as the primary information repository, existing tools can be used directly. Moreover, by having fragments represent the logical entities that the tools work with, it should be easy to modify existing tools to identify fragments and thus to set and access the fragment properties to more fully integrate them into the environment.

5.0 Fragment Scanning

One of the central issues in fragment integration is the identification of fragments and the information associated with them. In general, when a file is modified the information in the database needs to be updated. Similarly, when a tool that provides information to the fragment database is run, the corresponding database information needs to be updated. The central issues here are how to do the scanning, how to name fragments, and how to get other information into the database.

The first step in obtaining fragment information for a file is determining the type of file. The integration mechanism is designed to handle a variety of file types where each file type is defined as an object with associated methods. For a given file, the file type is determined by a set of heuristics that first considers the file's name extension (i.e. ".c" for C source), the results of the UNIX file command (which in turn has its own set of heuristics for determining file types), and, optionally, a method of the file type object that is passed the file contents to test.

Once a file's type is determined, we run an initial scanner on that file to identify the fragments. This scanner is designed to be as simple (and fast) as possible and to handle incorrect or incomplete files in a logical way. It is not a parser for the corresponding language per se. For example, our C and C++ scanners pay attention to block comments and sequences of blank lines as well as the syntax for functions and declarations. It totally ignores, however, the syntax of statements and expressions.

The language scanners for UNIX are complicated by the preprocessor. While we didn't feel that we had to accommodate arbitrary uses of macros and include files, we did want to handle all common uses and to limit the programmer as little as possible. In order to scan C or C++ we had to expand macros since macros are often used to define one or more functions and to define composite function names. In order to find all the macros we had to scan all the files included by the source file. Our scanner thus incorporates a simple implementation of the preprocessor. It scans include files only to find macro definitions and other include files. It does its own local expansion of macros. (This is needed in order to get accurate position information for fragments that are defined using macros.)

The other aspect of the preprocessor that complicates fragment scanning is conditional compilation. If we are interested in a particular instantiation of the system, we would know which conditions are applicable. However, because we are concerned with evolving the system in general, we have to assume that all conditions can apply for some version of the system and thus we want to extract all the code independent of conditional compilation. We do this in the scanner by keeping track of the implicit condition that has to hold for each fragment. This information is then passed on to the fragment database. We also keep track of the conditions that are implicit for each macro definition. This allows us to make an intelligent guess as to which instance of a multiply-defined macro should be used.

The output from this scanner is a list of the fragments in a file and the basic information associated with each fragment. This information includes the fragment's name and type, the start and end position of the fragment in the source file, and a hash value of the fragment's contents. The latter uses cryptographic algorithms so that any change in the contents are reflected in a change in the hash value. This can then be used to assign a date-last-modified to each fragment for doing intelligent system building or detecting changes [12].

Fragment positions identify the start and end position of the fragment in a file. This is generally easy to determine by keeping track of the start and end position of each token that is scanned and then noting the first and last token of each fragment. However, the start and end of a fragment must take into account comments and spaces that are not generally considered part of a programming language. The general problem of determining which token a comment applies to is quite difficult. We are dealing with a more limited case since we are only interested in comments for fragments. Our approach is twofold. First, the end position of a token is defined to include any blank spaces and comments up to the end of the line in which the token occurs. This associates end-of-line comments with the proper tokens. Second, we check for comment lines (lines that contain only a comment) that precede the start of a fragment and associate these with the fragment if there is not a series of 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 subsequent fragment.

Fragment names are designed so that the same fragment can be identified when a file is rescanned. This allows previous information associated with the fragment to be retained when the fragment is updated in the database. In addition, names have to allow us to track fragments as they migrate across files. Files are often split or merged and functions or declarations are moved around during software development and the integration mechanism must take this into account. Fragment names also must be unique so that the fragment database doesn't confuse two distinct fragments because they have similar names.

To handle this, we construct composite fragment names. Each name encodes the fragment type and a name derived from the context. In addition, names that are inherently local to a file, for example, file-local variables or functions, include the file name as part of the fragment name. For C and C++ we use the name being declared as the main component of the fragment name. This is complicated by C++'s use of overloading. Here we include the argument types of a function as part of the name. Any solution to naming is going to have pathological cases where fragments will not be correctly matched or are duplicated. Our solution attempts to avoid problems with duplicated names as much as possible. It also attempts to insure that correct matching occurs in most normal situations. Perhaps the most common situation we do not currently handle is that addition of an extra argument to a C++ function. We hope to gain experience with name mismatches over time and to eventually add heuristics to the fragment database to handle the more common cases. In addition, the output of each scanner is checked for duplicate fragment names within a file and such fragments are either merged or renamed to avoid duplicate names. This is designed to handle the case where there are two different conditional versions of the same function or declaration. The alternative is to encode the conditions as part of the fragment name.

The initial scanner is designed to only provide the basic information about fragments and not to provide lots of semantic information or other attributes. Our experience with FIELD is that such information is best provided directly by the tools that determine it, i.e. semantic information should come from the compiler, profiling information from the profiler, etc. This is accomplished by using additional scanners for a source file that have triggers other than the original file being updated. We get compiler information on Sun systems by scanning the source browser database that is created by the compiler. The database checks this information periodically to see if it has been updated and, if so, runs the corresponding scanner to update the semantic information for any files that have changed. The general framework allows arbitrary secondary scanners to be associated with a file type and provides a general mechanism for determining when these scanners should be run.

Attribute update in the fragment database can occur at different times. Some attributes, such as the date last modified for a fragment, are updated when the fragment is first scanned. Other attributes, such as references and definitions, are updated when the semantic scanner is run. Still other attributes, such as links between code fragments and requirements are created by the programmer and should only be updated by the programmer. To accommodate these different needs, the fragment database provides flexible attribute support. An attribute associated with a fragment consists of a value and a pointer to an attribute object. The attribute object names the attribute and describes its properties. Attributes are in general maintained when a fragment is updated. This allows semantic attributes and permanent links to be maintained as the user edits. Each attribute object identifies a set of scanners that generate the associated values. Whenever one of these scanners is run, the old values of the attribute are discarded and either the attribute is removed or the value is replaced with a default value specified by the attribute object. This is sufficient for maintaining links and for detecting dangling references after editing.

6.0 The Fragment Database

The key to making fragment integration a success is to provide a fast and simple database that is capable of meeting the needs of the various tools in the environment and of handling fragment information efficiently.

The first problem that must be solved is determining the domain of the fragment database. Ideally, there would be one fragment database for each project. Unfortunately, this does not work for real projects. Most projects are not self-contained. They use libraries or share components with other projects or they make use of other projects. It would be impractical and error-prone to attempt to duplicate fragment information for shared components. Instead, our implementation of fragment integration introduces a notion of a project and provides a separate fragment database for each project. Projects can use other projects in which case the database for the first project will query the database for the used projects as necessary.

A project is defined by giving it a name, associating a set of files and directories to identify its contents, and providing the set of projects it should refer to. If a directory is associated with a project, then all files that are in that directory or any of its subdirectories are considered in the project. If a binary file is associated with a project, then any source file that is used in building that binary is considered part of the projects. This definition of a project seems to handle most of the normal uses of the UNIX file system for program development.

Project objects are used as the front end to the fragment database associated with a project. Projects can be identified by providing the project name or by providing a file name to be checked. Once a project is identified, database queries can be sent to the project object and to be forwarded to the corresponding database. The actual database is run in a different process and is communicated with using the broadcast messaging facilities provided by FIELD.

The actual database system is an in-core relational query engine with object-oriented extensions, notably object identifiers and method calls. The database implementation provides a SQL-like query front end (it eliminates some of more arcane aspects of SQL and extends the language to deal with objects), and an algebraic implementation. The operators of the algebra can be dynamically defined as can optimization strategies and rules for these operators. This has allowed us to customize the implementation for its use as a fragment database. In particular, we have built-in operations to do symbol table lookup and dependency checking. This is modeled on previous work on extensible database systems [6,10].

The database implementation is multithreaded to allow multiple queries to be processed simultaneously. Each query request specifies a priority and these priorities are then associated with the query threads. This approach allows the database to be used for rapid informational queries such as finding the file, start, and end position of a fragment given its name or the value of attribute for a given fragment while still processing complex queries for other tools.

The database currently provides seven object sets or relations. Each relation consists of basic fields that are directly accessible through the database system and are stored when the database is saved, and computed fields. Computed fields are used for fast update, for implementing the built-in operations, and for caching computed information.

The **fragment** relation contains the basic fragment information: the name, hash value, date last modified, containing file name, containing fragment name, fragment type, and start and end positions. In addition, it contains compute links to the parent fragment and the file object, and a lists of the objects in other relations that are relevant to this fragment. The **file** relation contains information about a file. This includes the file name, the date last modified, and the language. It also contains a computed list of the fragments in the file.

The **ref** and **def** relations contain symbol table information for computing dependencies and following semantic links. Each includes the fragment name, the name that is referenced, the line number of the reference, the and the type of object if it is known. The ref relation also indicates if the reference is a read or a write to the name. Both relations contain a computed field pointing to the actual fragment. The ref relation also contains a computed field that is the corresponding definition object. This is computed on demand and cached in the object. Scoping information is contained in the **use** relation. This contains the parent and child fragments as well as the type of scope relationship that exists between these. Computed information here includes the actual source and target fragments.

The **attr** relation describes an attribute. This include the attribute name, the type of value associated with that attribute, and the rules to be used for updating the attribute. These rules determine when the values associated with the attribute should be discarded upon rescanning and what initial values should be used for the attribute. Attributes can be removed for when a particular scanner is run, whenever the fragment is updated, or never. The **prop** relation then contains the values for a given fragment and a given property.

The database objects are designed to both contain the necessary information and to support rapid updating of the information. One of the concerns with a database for a software development environment is that updates are generally more frequent than queries. The database has been designed so that all the fragments in a file can be updated in one operation when a scanner is run. This is done by finding all the fragments for the given file and marking these as unsourced. This may involve removing attribute values. Next, the initial fragment scanner for the file is run. As each fragment is identified and passed to the database, its name is matched against the previous fragments. If a matching fragment is found, it is updated with the new information.

As an example of the use of the fragment database in the environment, consider the problem of developing an intelligent configuration manager. The tool needs to know when a file needs to be recompiled. Since a modification date is maintained for each fragment, the database can be used to determine this on a much finer level than if the file has been changed. Thus a query like:

```
SELECT DISTINCT f.file
FROM fragment AS f, fragment AS g
WHERE g.modifyTime > time &
   EXISTS (
      SELECT * FROM def AS d, ref AS r
      WHERE r.fragment = f & r.def() = d & d.fragment = g )
```

would find all files that contain fragments that reference an item that is defined in a fragment that has a modification time greater than the specified time. This query shows our use of extended SQL in the method call *r.def()* which returns the object identifier for the definition object of the given reference, and in the comparison of this object identifier with the range variable *d*. As this query is one that would be asked frequently and as the dependency relation between fragments is more complex than simple definition-use (it must take into account scope inclusion, deletions of definitions so that what was previously defined is no longer available, and type definitions for items used) the actual implementation of the query would use the *dependsOn* method that gathered and cached the dependency information:

```
SELECT DISTINCT f.file
FROM fragment AS f, fragment AS g
WHERE g.modifyTime > time & f.dependsOn(g)
```

The fragment database communicates with other tools through the FIELD message server. It currently handles four messages. The first is a request to update itself. This causes all appropriate scanners to be run on all files that have changed. The second is to add a file to the database. This runs the initial scanner on the given file, and then determine what if any additional scanner should be run. If the file already exists in the database, it is explicitly updated. The third is used to set attributes of fragments. The fourth handles queries. It takes the query priority and a string containing the SQL query and returns a file name in a globally accessible area that contains the result of the query.

7.0 Conclusions

Fragment integration provides an alternative approach to obtaining the benefits of data integration in a software engineering environment without the excessive costs. It involves maintaining a database of program and system information that is lightweight and automatically updated using a collection of scanners. It involves maintaining references to portions of the original files that continue to be the primary repository of information.

This approach allows us to continue to use existing tools without any modification since the original files remain, and to extend these tools easily in a reliable way. We have already demonstrated in FIELD how a program database can be used to augment the editor with commands such as “go to the declaration of a selected item” or to provide information to a visualization front end.

Fragment integration can also be used to create new tools. The basic mechanisms support the extraction and replacement of fragments in a file. This allows us to create an editor that can be specialized to the task at hand. For example, if the user wishes to edit a particular function, the editor could display that function, declarations of the items that are used in the function, documentation for the function, test cases for the function, and fragments that contain calls to the function in a single view. Alternatively, the call sites could be brought up one at a time in an alternate view or test cases and their output could be shown in an auxiliary window as the function was updated. The basic idea here is to migrate from programming as a linear editing task to programming as the development of an electronic document.

Additional tools we are developing on top of fragment integration include an intelligent version of UNIX *make* that uses fragment-level dependencies rather than file-level ones; a debugger front end that uses a read-only display of fragments to show the items relevant to the current context; a version control system that supports version management of software units that can be fragments or logical combinations of fragments (i.e. files, directories, or related sets of fragments); and visualization tools to show the fragments in a system and their relationships.

8.0 References

1. 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).

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

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

4. 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).

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

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

7. 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).

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

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

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

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

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

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

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