Software Configuration Management as a Mechanism for Multidimensional Separation of Concerns

Mark C. Chu-Carroll
IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, USA
mcc@watson.ibm.com

Sara Sprenkle
Department of Computer Science
Levine Science Research Center
Duke University
Durham, NC, USA
sprenkle@cs.duke.edu

February 25, 2000

1 Introduction

Real software rarely conforms to one single view of the program structure; instead, software is sufficiently complex that the structure of the program is best understood as a collection of orthogonal divisions of the program into components. However, most software tools only recognize the decomposition of the program into source files, forcing the programmer to adopt one primary program decomposition which is well-suited to some tasks and poorly suited to others.

Software tools can overcome this weakness by allowing programmers to transform their view of the program to a structure which is more appropriate for the task they need to perform. We propose that a *software configuration management (SCM)* system, which stores the source code for the project, can perform this task. By providing the SCM system with the capability to generate orthogonal program organizations through compositions of program fragments, the SCM system can support orthogonal decompositions of the program without performing any automatic alteration of the source code.

To illustrate the importance of using different decompositions, consider the compiler, a classic example of a project with multiple orthogonal decompositions. The compiler's two more obvious decompositions—one structural, one functional—are illustrated architecturally in figure 2 A.

The structural decomposition is the interpreter design pattern, in which the system is decomposed into components (classes), where each component defines all the behaviors of one structural unit of the program. This decomposition is based on an inheritance hierarchy, which is in turn based on the syntactic structure of the language.

A functional view sees the compiler as a chain of components connected by data flowing between the components. The functional decomposition is based on this view: the parser generates an abstract syntax tree (AST) from the source code. The analyzer performs a pass over the AST, decorating nodes with semantic information. The intermediate code generator creates a stream of intermediate code instructions during its pass over the decorated AST. The optimizer operates on the intermediate code stream and generates a new optimized stream. Finally, the code generator translates the optimized intermediate code into executable binary code. In the functional decomposition, each of these components is provided by a component of the system.

Given conventional tools, these two decompositions are mutually exclusive. The code can be written in the interpreter pattern, but then the functional decomposition is not available. When faced with a programming task well-suited to the functional decomposition (for instance, replacing the type analyzer with a better type inference algorithm), the programmer needs to view and manipulate code scattered through dozens of different files. The overall operation of the functional components is obscured, and changes to a program's functionality become significantly more difficult.

Adopting the functional decomposition rather than the structural decomposition makes replacing functional components easier but produces its own set of problems. For instance, in the functional decomposition, adding a new syntactic structure to the language requires changing all of the components of each functional decomposition, whereas it only requires adding a single new component to the structural decomposition.

The solution to this problem is to build new tools that allow programs to be organized in multiple dimensions: that is, to categorize the code in multiple ways, each of which produces a distinct decomposition of the program. Each of these dimensional decompositions represents a view of the program according to some organizational principle.

We believe that there are three primary approaches that tools can take to provide this kind of functionality:

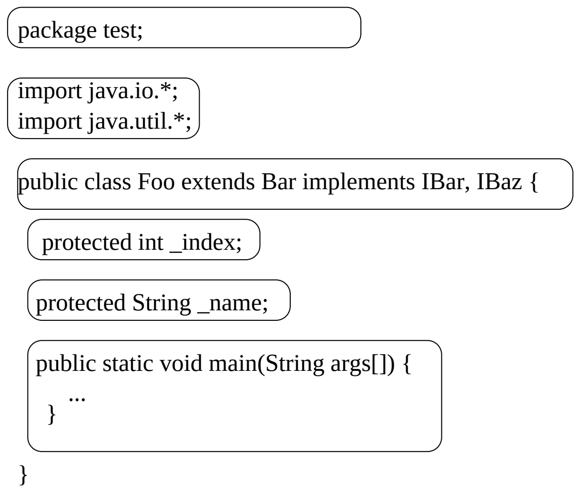
1. The transformational approach, in which an automated tool transforms code from a canonical program organization into some other organization;
2. The compositional approach, in which the distinct concerns that make up a system are implemented separately, and are then composed into a complete system according to some composition rules;
3. The query-based repository approach, in which the code is not transformed or composed, but withdrawn from a repository in organizations which correspond to different program decompositions.

We believe that tools that are based on program transformation and composition, while powerful, introduce complexity to the programming process, and are likely to introduce confusion and errors. Rather than transforming the program, we believe that providing tools that simply re-organize the code, without making *any* semantic changes of any kind can provide the necessary support, without introducing the complexity of composition or transformation. Since most programming projects use a software configuration management system as a central repository for storing their code, we believe that the appropriate way to implement this functionality is be integrating it into the SCM system.

We are building such an SCM system called Coven, which provides support for multidimensional program organization. Coven is tightly integrated with a programming environment to provide strong support for it novel features. In the rest of this paper, we will present the relevant features of Coven as an illustration of how such features can be provided by an integrated SCM system.

2 Using SCM for Multidimensional Separation of Concerns

Our approach to the multiple decomposition problem is to avoid program transformation. We propose a simpler solution that uses queries to extract program fragments of a particular dimension of concerns from a program repository to gain the advantages of multidimensional separation. Since most software developers use an SCM system to store their source code, we have focused on integrating the query support with an advanced configuration management system.



```
package test;

import java.io.*;
import java.util.*;

public class Foo extends Bar implements IBar, IBaz {

  protected int _index;

  protected String _name;

  public static void main(String args[]) {
      ...
   }

}
```

Figure 1: Decomposition of Java source into versioned program fragments

2.1 The Coven Repository

The Coven repository is a change package-based version store with four key features:

1. Versioning is performed on the smallest independent fragments of program source in the source language. For example, in Java, fragments are individual method/field declarations. The division of a standard Java source file into fragments is illustrated in figure 1. Within the repository, fragments are stored as text with the ability to rapidly parse that text into a simple AST. We call this *fragment-based versioning*.

2. Fragments in the repository have a list of associated properties that allow the user of the repository to associate metadata with the fragments. Programmers can then associate fragments with user-provided information. For instance, a programmer could mark a fragment as being an element of a particular named component of the system.

3. Fragments are retrieved from the repository using queries that ogenerate *virtual source files (VSFs)*. Virtual source files encapsulate slices through the full set of fragments that embody some modularization of the program.

4. Change packages are provided in terms of change sets that create *consistent project versions.* While details of project consistency and what it means in our system are beyond the scope of this paper, project consistency is a mechanism that ensures that change sets are integrated into the repository in a manner that ensures that a fully consistent system, identical to the

```
Example 1:  Extracting all fragments that either
implement analyzeType, or that implementations
of analyze type depend on.
(all fragment (from compiler)
   (where
      (impls fragment ’analyzeType)
      (exists (otherfragment
            (impls otherfragment ’analyzeType))
         (depends otherfragment fragment))))

Example 2:  Extracting a source file for a class
(all frag (from compiler)
   (format java-source-file)
   (where
      (inClass frag ’ArithmeticExpression)))
```

Figure 3: Example queries to generate virtual source organizations.

state of a previous complete system, can always be properly reproduced.

The combination of these four features are the basis of Coven’s flexibility and adaptability. Fragment-based versioning decomposes programs into smaller elements, and, with the query-based retrieval mechanism, these smaller elements can by assembled in arbitrary ways to form source organizations that correspond to different dimensional decompositions of the program.

For example, returning to the compiler project, the two primary dimensions that we discussed (structural and functional) can each be represented by a different virtual source organization. In figure 2, we illustrate these two decompositions in subfigures a and b, and in subfigure c, we show the source slices that correspond to the two decompositions. The structural decomposition is shown as the vertical slices, which generate the standard class-based structural organization of a compiler written in an object-oriented language. The functional decomposition is depicted by the horizontal slices that extract the methods and fields that are elements of each of the functional components.

2.2 The Coven Query Language

To take advantage of the fragment-based versioning provided by Coven, programmers need a flexible interface they can use to build new project configurations Coven’s query language gives programmers the ability to customize source code organizations by specifying how to select and assemble components into a virtual program organization. In the rest of this section, we will describe the subset of the query language that allows programmers to generate new virtual source organizations.

Using the query language, programmers can create a collection of queries, each of which generates one virtual

```
(collection
    (source "typeAnalyzer" (all ...))
    (source "codeGenerator" (all ...)))
```

Figure 4: Collecting queries to form a virtual source organization

source file. A query specifies a project from which fragments should be drawn and a condition that a fragment must satisfy to be included in the VSF. Individual queries are joined together into collections to specify a complete virtual program organization.

The query language itself is implemented in an extensible fashion, consisting of a collection of language-independent standard clauses and an extension mechanism that allows programmers to add new clauses to the language. Possible new clauses could be project, language, or domain-specific.

We illustrate a few simple queries in figure 3 and how they can be assembled to form virtual source organizations in figure 4. Example 1 draws fragments from the project `compiler`. Each fragment must either implement the method `analyzeType` or be required by another fragment that implements `analyzeType`. That condition is specified by two query clauses—one for each of the two possibilities—joined by an implicit logical *or*.

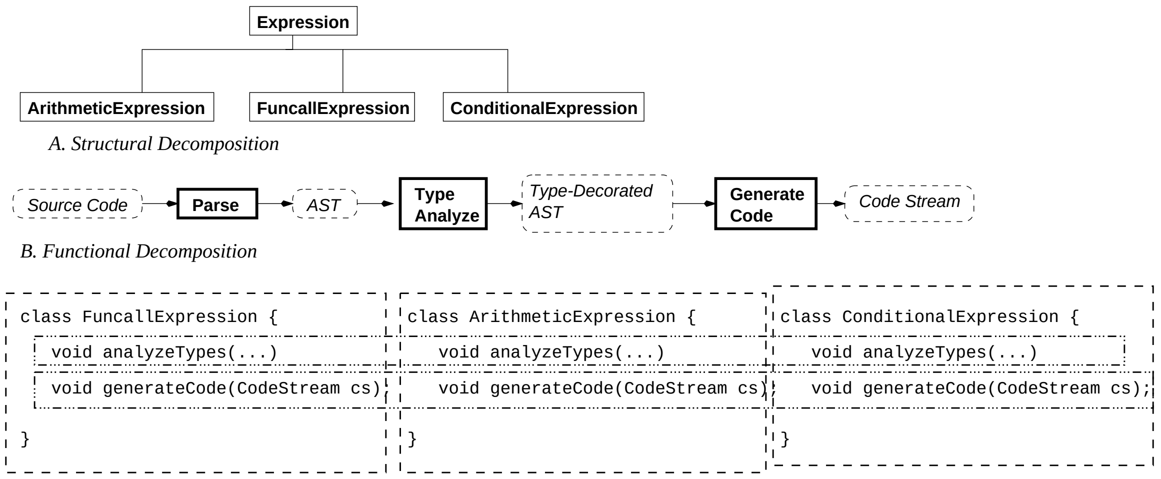
Example 2 is a query that will extract a VSF containing the fragments that make up a particular Java class. The `format` clause means that the resulting VSF should be formatted as a Java class file. In general, the `format` clause extracts program source from the repository in formats that are usable by other, non-Coven specific tools. In particular, this query can extract a VSF that could be compiled by a standard Java compiler. The set of formats that can be generated by a query using the format clause is extensible.

Coven’s dynamic query and composition support can also be used to manage different versions or configurations of a project that support different feature sets. Since queries dynamically generate the source code that is compiled, they can specify the set of features or versions that should be included in a particular source file.

2.3 Manitoba: The Coven User Interface

The use of an SCM system like Coven introduces added complexity to the task of the programmer. The programmer gains expressiveness and organizational flexibility at the cost of dealing with queries, consistency, and the loss of intrinsic context. To hide the added complexity, a programming environment can make the manipulation of multiple program organizations more tractable.

We have built a programming environment called Man-



```
class FuncallExpression {
  void analyzeTypes(...)
  void generateCode(CodeStream cs)
}

class ArithmeticExpression {
  void analyzeTypes(...)
  void generateCode(CodeStream cs)
}

class ConditionalExpression {
  void analyzeTypes(...)
  void generateCode(CodeStream cs);
}
```

A. Structural Decomposition

B. Functional Decomposition

C. Source slices corresponding to virtual source organizations for different decompositions

Figure 2: Two decompositions and the corresponding organizational slices

itoba, which allows the easy integration of external tools into a flexible linked-pane style programming environment. The environment provides tools that simplify the generation of new queries, as well as a variety of facilities that make it fast and easy to use standard queries.

For instance, consider a scenario where a programmer is working on our compiler. The programmer needs to make a change to the type analyzer in the functional decomposition and edits the type analyzer component. While working on the `ArithmeticExpression` type analyzer method, she realizes that she does not know some properties of the analyzer's compilation context. She clicks the right mouse button for a menu. Selecting the menu option `open class view` opens a new window with the `ArithmeticExpression` class presented in context.

3 Related Work

The idea of multiple program views based on rearranging source organizations has been explored. The Gwydion project from CMU[4] built a hypercode programming environment called *Sheets*. Like Coven, the Sheets system subdivides code into program fragments and allows fragments to be dynamically assembled into new virtual source files, which they call sheets. They provide a mechanism for creating *hyperlinks* between sheets that provides a capability similar to our artifact associations. The Sheets environment used a query language to generate sheets viewed by their UI. However, the Sheets system did not integrate SCM with this dynamic view support. Further, this view system was based on their use of a repository which could only be accessed from within their system.

The Subject Oriented Programming/Hyperspaces[1, 2] project at IBM has also explored alternative program organizations. Their focus has been on generating new program organizations by performing program source transformation. This kind of program reorganization is a heavyweight process geared towards the needs of software maintenance and is expected to be used very rarely.

IBM's VisualAge Smalltalk includes a fragment-based program repository called ENVY[3]. ENVY does not provide any facility for allowing programmers to generate queries to extract code from this repository. It provides a standard collection of "views" that can be extracted from the repository, corresponding to a single, canonical file-based decomposition of the program.

4 Conclusions

We presented an alternative mechanism for allowing multidimensional separation of concerns. Instead of using a tool that performs source code transformation or composition, we use a flexible configuration management system in conjunction with an integrated programming environment to provide functionality similar to transformational tools. Our system is based on the use of fragment-based versioning and flexible views built from collections of fragments to allow programmers to dynamically generate new organizations of the program that correspond to some dimension of concern.

References

[1] OSSHER, H., AND HARRISON, W. Combination of inheritance heirarchies. In *OOPSLA* (1992), pp. 25–40.

[2] OSSHER, H., KAPLAN, M., HARRISON, W., KATZ, A. E., AND KRUSKAL, V. Subject-oriented composition rules. In *OOPSLA* (1995), pp. 235–250.

[3] OTI. ENVY/Developer: The collaborative component development environment for IBM visualage and object-share, inc. visualworks. Webpage: available online at: “http://www.oti.com/briefs/ed/edbrief5i.htm”.

[4] STOCKTON, R., AND KRAMER, N. The sheets hypercode editor. Tech. Rep. 0820, CMU Department of Computer Science, 1997.