Supporting Aggregation in Fine Grained Software Configuration Management

Mark C. Chu-Carroll
mcc@watson.ibm.com

James Wright
jwright@watson.ibm.com

David Shields
shieldsd@us.ibm.com

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10522

ABSTRACT

Fine-grained software configuration management offers substantial benefits for large-scale collaborative software development, enabling a variety of interesting and useful features including complexity management, support for aspect-oriented software development, and support for communication and coordination within software engineering teams, as described in [4]. However, fine granularity by itself is not sufficient to achieve these benefits. Most of the benefits of fine granularity result from the ability to combine fine-grained artifacts in various ways: supporting multiple overlapping organizations of program source by combining fine-grained artifacts into virtual source files *(VSFs)*; supporting coordination by allowing developers to precisely mark the set of artifacts affected by a change; associating products from different phases of the development process; etc.

In this paper, we describe how a general *aggregation* mechanism can be used to support the various functionality enabled by fine grained SCM. We present a set of requirements that an aggregation facility must provide in order to yield these benefits, and we provide a description of the implementation of such an aggregation system in our experimental SCM system.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Evolution—*Version Control*; D.2.9 [**Software Engineering**]: Management—*Software Configuration Management*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments—*Programmers Workbench*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software Selection*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models, search process, selection process*

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

General Terms

Languages, Management

Keywords

Fine grained storage, aggregation, dynamic program organization

1. WHY? MOTIVATING AGGREGATION

Managing software artifacts in a software configuration management system is a fundamental part of modern software engineering practices. Most SCM systems manage software artifacts in terms of files, which are relatively coarse grained entities. Adopting a finer granularity such as individual methods or functions offers substantial benefits for large-scale collaborative software development, particularly in the areas of complexity management and communication/coordination support for software engineering teams, as described in [4]. However, an SCM tool cannot produce these benefits simply by adopting finer storage granularity. Most of the benefits of fine granularity result from the ability to combine fine-grained artifacts in various ways: supporting multiple overlapping organizations of program source by combining fine-grained artifacts into virtual files; supporting coordination by allowing developers to precisely mark the set of artifacts affected by a change; associating products from different phases of the development process, etc.

In the context of an SCM system, aggregation is a facility to allow the creation of versioned objects formed from collections of other objects. For example, in a typical SCM system, a directory is an aggregate object formed from a collection of file objects. In a fine-grained system, aggregates can be used to represent a wide variety of interesting relationships between collections of versioned artifacts. For example, in figure 1, we illustrate two overlapping aggregates: one representing a relationship between a UML interaction diagram and the code artifacts that are participants in the interaction; and the other representing a set of Z schema artifacts and the code implementing the operations specified by the schema.

Fine granularity without aggregation support does have value. With file-based SCM, all repository facilities (from locks to version histories) operate at the file level and above. In these systems, it is hard to assess the cost of integrating change-sets, file locking causes unnecessary bottlenecks when several developers need access to different sections of a given source file, and change histories for individual methods

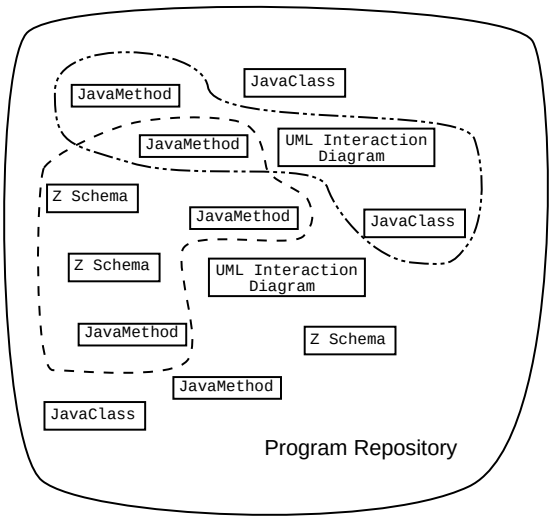


Figure 1: Aggregates representing versionable cross-cutting relationships

and functions are difficult to generate (especially if methods migrate between files over time). Fine-grained SCM can help address all of these problems.

However, fine-grained SCM without aggregation also introduces substantial complexities into a system. Experimental data gathered using our current implementation indicates that changing from file to method level granularity increases the number of managed artifacts by one to two orders of magnitude. To make fine granularity manageable, especially given the resulting increase in the number of artifacts, both the SCM repository and related development tooling must provide mechanisms that support sophisticated aggregation functionality.

Our solution for these issues is based on the comprehensive use of aggregates as first-order entities. We claim that such reified aggregates enable a host of useful capabilities that extend well beyond simply making SCM work better, including:

- Multidimensional program organization,
- Support for distributed development,
- Linkage and relationship management for heterogeneous artifacts,
- SCM support for fragment-oriented technologies such as aspect-oriented programming,
- Improved coordination support for collaborative development, and
- Improved lifecycle and process support.

Details on how these features can be provided by a fine-grained SCM system can be found in [4, 5]. In this paper, we will describe the details of the aggregation system needed to implement them.

2. WHAT? AGGREGATE REQUIREMENTS

As discussed earlier, aggregate support is required for many distinct purposes. In order to properly support aggregation within an SCM system, the aggregation mechanisms must be carefully designed with all of these purposes in mind. We have analyzed various applications of aggregation in fine-grained SCM and produced a set of requirements for aggregate typing and generation and support for legacy source code and tooling.

2.1 Aggregate Typing

Aggregates are used for purposes ranging from representing source-file-like collections of source code fragments to providing high-level semantic interlinkages between artifacts of many different kinds. The common thread between these varied usages is that all rely on reifying a relationship between other artifacts as a first class artifact in its own right. This basic principle dictates two primary requirements:

First and foremost, the aggregate system must support a type mechanism that allows users to define relationships, to define the types of artifacts that can fill a particular role within a relationship, and to distinguish between aggregates representing relationships of different types. ([**?**] has an excellent discussion of role modeling and OOD.)

Second, the aggregate system needs to provide an extensible mechanism for defining how a given aggregate type interacts with the versioning functionality of the repository. In particular, the system must provide support for maintaining the identity of the aggregate representing a particular relationship, and for defining how to handle a given aggregate when overlapping changes are merged.

2.2 Aggregate Generation

The set of aggregates managed by the system is highly dynamic, so there must be support allowing both developers and tools to create and destroy aggregates as needed.

Based on published experiments with similar fine-grained systems[25], it has been observed that developers frequently create aggregates as an exploratory tool, and that most of these are transient artifacts that will be discarded; however, users often do not know whether a particular aggregate will be valuable enough to preserve. Therefore, we believe that the mechanism for aggregate creation should have the following properties:

1. The aggregate system must support both transient and persistent aggregates;
2. The system must be able to generate transient aggregates extremely quickly;
3. All aggregates should be created as transients;
4. It must be possible to rapidly and simply convert transient aggregates into persistent artifacts.

Furthermore, to take full advantage of the aggregate system, and to support dynamic creation of interesting aggregates, the SCM system needs some level of knowledge of the semantics of atomic artifact types. For example, to create an aggregate showing all of the methods related to a particular program aspect, the system requires semantic knowledge in order to identify code matching a predicate specifying the aspect of interest.

2.3 Externalization

Fine-grained program repositories provide a dramatically different interface to software artifacts than that expected by traditional programming tools. Programmers cannot be expected to simply abandon their familiar editors, compilers, debuggers, and profilers; further, it is unreasonable to expect that all future tools will be implemented within the fine-grained model. Therefore, we believe that the aggregate system must provide support for generating external forms of aggregate artifacts suitable for use by other software development tools, and importing and extracting aggregates from output produced by such tools.

These externalization and internalization mechanisms must be integrated with the aggregate type system so that different types of aggregates can each support external formats that are appropriate for their contents.

3. HOW? AGGREGATION IN STELLATION

We have been building an experimental SCM system called Stellation, which is based on the use of fine (method-level) storage granularity. For Stellation, we have designed an aggregate system meeting the requirements identified. The system is based on a powerful typing mechanism for aggregate objects, combined with an efficient search technique that allows aggregates to be generated quickly and easily, and with externalization support based on standard mechanisms.

3.1 Aggregate Typing in Stellation

The Stellation aggregate type system is a simple system, resembling structure definitions in a language like ML, with a defined aggregate merge algebra (An overview of the notion of merge algebra can be found in [27] and [16]). An aggregate type is a structure consisting of named fields, each of which has a type. In addition, each field may be annotated by a *merge descriptor*, which defines how that field will be handled during a merge.

We will describe the type system in two parts. First, we will describe the basic types and the mechanisms for defining new types. Then, we will describe how annotations allow developers to easily define merge operations that make sense for their aggregate types.

3.1.1 Basic Types and Type Definitions

An aggregate type is similar to a structure type in a programming language: it defines a collection of named fields, each of which has a type. The types for a field can be a primitive atomic type, a semantic atomic type, a collection type, or a previously defined aggregate type.

The primitive atomic types are *Integer*, *String* (single line text values), *Text* (arbitrary length text values), and *Data* (arbitrary length binary data).

Semantic types are language dependent types added to the system using extension components. Each semantic type is a text or data artifact tagged with a constructor associating that semantic type with a particular semantic element of a programming language. For example, for Java we have semantic types for package declarations, imports, class/interface declarations and class members.

The built-in collection types are sets (for unordered collections), and lists (for ordered collections). In addition, an aggregate field can use the type of any previously defined aggregate type.

```
aggregate java_class {                                    (a)
   name :  [conflict] String
   package :  [conflict] java_package_decl
   imports :  [union] java_import_decl
   decl :  [conflict] java_class_decl
   members :  [linear] java_member List
}

aggregate java_viewpoint {                                (b)
   name :  [conflict] String
   description :  [linear] Text
   members :  [dynamic] java_member List
}

aggregate test_case { ...  }                              (c)

aggregate bug_report {                                    (d)
   title :  [conflict] String
   severity :  [largest] Integer
   description :  [conflict] Text
   subject_code :  [dynamic] java_member_decl Set
   test_data :  [union] test_case Set
}

aggregate specifies_relationship {                        (e)
   specification :  [union]Z_fragment Set
   implementation :  [union]java_member_decl Set
}
```

Figure 2: Sample Aggregate Declarations

Figure 2 presents some examples of aggregate usage and related aggregate type definitions with merge annotations (described in the following section). (a) shows how the aggregate mechanism can associate the set of artifacts comprising a Java class declaration. (b) represents a Java viewpoint. (c) and (d) show how an aggregate can associate a bug report with a set of subject code and test data. (e) illustrates how aggregates can represent relationships between heterogeneous artifacts — in this case, the relationship between a set of program specifications and the source code that implements the specified operations.

This type system provides a flexible way of defining new kinds of aggregates that is adequate for expressing all of the artifact types required by our aggregation facilities. It has the useful property that types can *never* be cyclic (no type declaration can reference any type that is not completely defined), which supports clear, simple merge operations for aggregates, as described in the following section.

3.1.2 Merge Annotations

Proper support for versioning of aggregates requires a mechanism for merging concurrent changes to a particular aggregate. The purpose of this system is *not* to define a perfect mechanism for merging changes: such a mechanism is an impossibility. The intention is to provide an easy to use mechanism that produces a good approximation of a valid merge result, which developers then examine and modify to produce the final result.

Our solution for defining merge operations is to allow developers to define field-wise merge behaviors in terms of simple *merge annotations*. Each merge annotation is either equivalent to a merge behavior definition in a traditional merge algebra, or defines an easily computed value in terms

Annotation	Applicability	Result
Base	Generic	b
Conflict	Generic	signal error
Latest	Generic	most recent of δ_1 and δ_2
Largest	Integer	$\max(\delta_1, \delta_2)$
Smallest	Integer	$\min(\delta_1, \delta_2)$
Linear	Text or List	common merge
Intersect	Set	$\delta_1 \cap \delta_2$
Union	Set	$\delta_1 \cup \delta_2$
DeltaSum	Set	$(\delta_1 \cap \delta_2) \cup (\delta_1 - b) \cup (\delta_2 - b)$
Dynamic	Set or List	result of dynamic query

Table 1: Aggregate Merge Annotations

of the different cases entering the merge.

A merge operation is defined in terms of three versions of an artifact: a base aggregate, and two modified artifacts derived from the base. We enumerate and briefly define the meanings of the annotations in Table 1. In the definitions, b refers to the base version; δ_1 refers to one of the modified versions, and δ_2 refers to the other modified version. A merge operation has two possible outcomes: a predicted result value, and *conflict*, which is an indication that there were concurrent changes that the system could not even attempt to reconcile.

The linear merge in the table refers to a common merge algebra based algorithm used to merge text files in most SCM systems. We generalize this algorithm to work not only on text merges, but on general ordered lists of values.

The *DeltaSum* merge operation is the set equivalent of the linear algorithm for ordered lists. It first removes anything that was removed in either delta, and then adds anything that was added in either delta. We call it deltasum because, intuitively, it is the result of adding the changes from the two deltas.

The *dynamic* merge annotation indicates that the collection value of the field is computed using a dynamic query. In this case, when an aggregate instance is created, a query is associated with the field. When two versions of the aggregate are merged, the field value is generated by recomputing the value of the query. (We introduce dynamic queries in section 3.2.) For dynamic merge, the merge result should be generated by re-evaluating the query *only* if the query expressions in δ_1 and δ_2 are the same; if not, we signal a conflict. Dynamic merges are executed in type-definition order, ensuring that the values a query operates on are determined before that query is executed. (For a type T to contain a field of type S, it must be true that S was defined before T; therefore, the values of artifacts of type S will have been resolved before any artifacts of type T are merged.)

We selected this set of merge annotations with two criteria in mind. First, we wanted to support merges for aggregate fields that behaved as much as possible like the text merges developers are familiar with (linear, deltasum); second, we wanted to provide operators capable of merging semantic aggregates without generating conflicts (union, intersection, and again deltasum).

Our approach has two fundamental advantages over existing file-granularity merge algorithms:

1. Methods are atomic, therefore changes and merges are presented in terms of semantically self-contained chunks with clearly delimited context. There will never be conflicts which cross method boundaries.
2. Many conflicts can be resolved automatically, and those that cannot are often easier to deal with.

Conflicts frequently occur because of simple rearrangements within a file or the addition or deletion of methods. Because our system manages and presents source as aggregates containing semantic fragments, the addition, deletion or relocation of a method is shown as exactly what it is and can often be resolved automatically. By contrast, when such changes are viewed as line-bounded difference regions (as with file-granularity merges), the loss of semantic context makes such changes difficult to understand and conflicts difficult to resolve.

Some examples of aggregate definitions with merge annotations are shown in Figure 2.

3.2 Dynamic Aggregate Generation

As discussed earlier, we claim that for an SCM system to fully deliver the benefits of fine granularity, that system must support some mechanism by which users can easily and rapidly search the repository for fragments that are likely elements of an interesting aggregate, and then use those search results either transiently or as a persistent versioned artifact.

In Stellation, the key to aggregate creation is a query language. When the user wants to create an aggregate, she is presented with a form. The form allows the user to enter the values of fields, either as literal values (with UI assistance to identify candidate for field values), or as dynamic expressions that will be evaluated to produce the value of the field.

In both cases, the developer will frequently use a query mechanism to search the repository for relevant fragments. If a field has a static artifact or artifact collection value, the developer can use the query language to identify a set of candidate artifacts, which can then be used to set the value of the field. If the field is dynamic, the developer provides a query which will be evaluated whenever a new version of the aggregate is generated.

3.2.1 The Stellation Query Language

The Stellation query language was designed with the following goals in mind:

1. **Expressiveness.** Users should be able to express reasonably complex predicates in order to generate the appropriate set of aggregates.
2. **Ease of use.** The query language should be easy to use and understand, and should be based on familiar underlying concepts.
3. **Efficiency.** It should be possible to evaluate a query and return the result quickly.
4. **Incrementality.** Once a query result is complete, it should be easy to incrementally refine that result, by modifying the query predicate and/or manually adding and removing artifacts from the result.
5. **Extensibility.** It should be possible to add extensions to the query language, and to allow the query engine to be extended in ways that allow query extensions to be executed as quickly as built-ins.

We have developed a language loosely based on the idea of set comprehensions in a typed set theory. A query expression describes a set of program artifacts which should

be included, and may contain nested quantifiers and subqueries.

```
v is a java_viewpoint aggregate
s is a specifies_relationship aggregate

(1) Populate a java_viewpoint aggregate
v.name = "type analysis"
v.description = 'code that analyzes types'
v.members = all x :  java_member |
   x.name = 'analyzeTypes' OR
   (exists y :  java_member |
   y.name = "analyzeTypes" AND y dependsOn x)

(2) Adding a specific fragment to a query:
v.members = v.members UNION { artifact(id=297) }

(3) Populate a specifies_relationship for artifacts related
to a pre-selected schema S582. Assumes developer is
using UI to mark fragments which implement specs via
their "implements" property
s.specification = { S582 } UNION
   (all x :  Z_fragment| references(S582, x)
s.implementation = all x :  java_member |
   x.implements = S582
```

Figure 3: Pseudocode examples using queries to populate aggregates

The query language syntax is illustrated in Figure 3. The set of types and predicates available depends on the programming language extensions loaded by the system. The query syntax shown is used by developers, but the field assignment syntax is not; rather, developers will typically use a UI allowing them to build aggregates and queries using both manually entered query clauses, and clauses generated automatically by the system. For instance, in query (1), the dependency clause would not be entered manually, but would be appended by a UI action.

The query language structure makes it simple to augment queries with clauses corresponding to the kinds of incremental updates that occur in typical systems. This is illustrated by Figure 3, Example 2.

3.2.2 Implementing Queries Efficiently

The Stellation repository typically contains a large-to-very-large number of artifacts. Because we use a finer artifact granularity than file-based SCM systems, we typically store almost two orders of magnitude more artifacts for a given project, compared to a file-based system. However, our system is also designed so that developers view code through dynamically created aggregates. It is therefore crucial to provide prompt query evaluation, even for a developer's workspace containing an extremely large number of artifacts.

Furthermore, like most SCM systems, Stellation stores all artifacts in a central repository;and programmers typically view a subset of their system in a workspace. In such an environment, performing global analysis is extremely expensive. In order for a search technique to meet our performance requirements, analysis must be local and incremental.

To this end, we employ a query engine that incorporates information retrieval (IR) techniques. Our approach is loosely based on the keyword vector method of searching document libraries, introduced by Gerald Salton in the SMART system [24].

The keyword vector solution is an early IR technique based on selecting a set of *keywords* likely to identify the subject matter of a document. To generate an index, one searches the document library and records the frequency of each keyword in each document. For each document, a vector is generated; each keyword is assigned a vector index. Thus, each document is represented by a sparse multidimensional vector defined by its contents.

Each query is likewise analyzed for the keywords it contains, and a query vector is generated. The system then searches the library by computing the cosine of the query vector against each document vector. The result is known as the *cosine score* for a document. Any document with a cosine score within a given distance from 1 is considered as a potential match. This mechanism typically creates an extremely small candidate set from a large initial document set; the candidate set is then examined further to produce the query result.

Our approach is based on this technique, modified to take advantage of program code semantics. In classic IR, the search texts are natural language documents, for which precise semantic information is difficult or impossible to generate. However, when the search texts are program source in a given programming language, it is simple to extract and use contextual semantic information. We apply this information in three ways.

First, we can rely on precise matching; cosine scoring is not necessary. With program source, both query and summaries can be expressed precisely, without the need to handle ambiguity. Second, in a program artifact, the set of relevant index keywords is the set of non-local identifiers referenced by the fragment. Finally, for program source, it is possible to enumerate the specific usage of each referenced identifier, and not simply its presence.

Applying these properties of program artifacts, we replace the keyword vector with what we call a *usage summary*. This comprises a list of the non-local identifiers contained in the artifact, annotating each member with usage context tags. (The analysis to generate a usage summary and the precise set of usage tags is programming language dependent, and is provided in our system by an extension component.) As we will show in section 3.4.2, query performance with this approach is much faster than with a conventional IR approach.

Each query produces a *candidate set* of artifacts that are potential elements of the resulting collection. For some queries (e.g. queries with nested existentials), further analysis may be needed to determine if the candidate elements are proper members of the result set.

For example, our Java component analyzes Java artifacts for the following usage contexts, where x denotes an identifier: *Declaration* (artifact contains a declaration of x), *Assignment* (artifact contains an assignment to x), *Use* (artifact uses the value x), *Call* (artifact contains a call to a method named by x), *Target* (artifact contains a call to a method where x is the target of the invocation), *Inherit* (artifact contains an inheritance clause naming x).
Figure 4 gives a small example:

Stellation currently performs queries against a *workspace*: a collection of program artifacts from the repository that include *at most one* version of each managed artifact. The workspace can access a table containing the full set of artifact identifiers and usage summaries.

The first step in query evaluation is normalization: we require all queries to be in disjunctive normal form. This is done using standard algorithms. Once the query is in DNF, we generate a summary string for each conjunctive sub-clause of the DNF query. Generating a summary string is trivial: each usage class in the summary has an associated query language predicate, and each reference to a predicate generates an entry in the summary string.

For example, given the query "`all x : JavaMember | x defines foo AND x assigns bar`", the system would generate a query summary of "{ `foo`*(assigned)*,`bar`*(assigned)* }".

Thus, for each query we get a set of query summaries, one for each conjunctive clause in the DNF. We compare these against the summary strings for each workspace artifact, using IR vectors for the query expression and candidate artifacts. The IR vector for each candidate is then collapsed to the dimensionality of the query. If the two resulting vectors are identical, there is a potential match. All such matches are then verified to produce the result.

An evaluation of the performance of this query mechanism for aggregate generation is provided in section *3.4.2*.

3.3 Aggregate Externalization

Externalization is the capability to take an aggregate artifact within a Stellation repository and translate it into a form which can be used by external tools. There are two key pieces to this process: *export* (take an aggregate within the repository and translate it into its external form), and *import* (take an externalized aggregate, and translate that back into an internal form, generating a new version if the aggregate was modified).

Our approach to externalization is based on the use of standard XML tools. The XML community has done extensive work on document transformation using XSLT[8] and XML formatting objects[21]. We export all aggregates into an XML format defined by an aggregate schema, and then allow developers to use any XML tool they prefer to translate the resulting document into a desired format. Import is handled similarly; developers use external tools to translate their external form back into the Stellation XML schema, and the resulting document is reintegrated into the repository. We plan to provide import/export tooling for several common formats.

We expect that, to make round-trip operations work with non-XML external formats, developers will use external forms that include marker tags identifying the boundaries between areas that should be translated into XML elements. Figure 5 presents one such external format. With these markers, it is easy to write a script that translates from this format to a simple XML format, which can be transformed back into Stellation aggregate form using XSLT tools.

```
public void foo() {
  x.bar(y);
  int k = z;
  bar(12, k);
  bim(x);
  x = bim(k);
}
```

Usage Summary: { `bar`(*invoked*), `bim`(*invoked*), `foo`(*declared*), `x`(*assigned,passed*), `y`(*passed*), `z`(*used*) }

Figure 4: A simple code artifact and its usage summary

3.4 Evaluation

Evaluating Stellation's aggregate support is difficult without extensive user studies, which we have not yet had the opportunity to perform. However, we can informally evaluate our system based on two criteria: the flexibility of the aggregate mechanism applied to a variety of tasks, and the performance of the aggregate generation mechanism on those tasks.

3.4.1 Aggregate Flexibility

We assessed the flexibility of the aggregate system by examining a set of three moderate sized software projects implemented in Java: JEdit[**?**] (a developers editor), Jakarta Ant[**?**] (a build tool), and a distributed programming environment developed at IBM Research called Manitoba. We identified a set of tasks that we believe are typical uses for aggregate facilities: program understanding, concern/aspect identification, problem management, and linking specifications with program code.

We then considered how to conduct each task using Stellation. Overall, our assessment is positive. For each task area, it was easy to assemble aggregates that did an adequate job. However, we did find that for some tasks, our ability to easily produce uniform models was limited. We will briefly describe our use and assessment of Stellation's aggregate facilities for each task area.

- **Program Understanding and Concern Identification.** Here, we first selected an aspect of a subject system that we wished to understand better, and then generated transient aggregates using queries specified to identify program artifacts likely to be related to that aspect. For example, the menu generation process in Manitoba is performed using a loosely coupled contribution mechanism. Using the Stellation aggregation generation mechanism, we produced an excellent approximation of the body of code involved in menu contributions.

- **Problem management.** In this case, our objective was to take selected bugs, and, for each, generate an aggregate containing data for test runs that produced the bug, along with a collection of code containing likely locations for the bug. Our experience here was mixed. On the positive side, our aggregation mechanisms did provide a good way of selecting and manipulating candidate program fragments for the bug. On the negative side, we concluded that Stellation's type mechanisms were overly strict for this task. Sets of test data are not homogeneous; we cannot build a single aggregate containing collections of heterogeneous type data while maintaining the ability to identify the types of that test data.

- **Linkage of code and specification.** In this case, our objective was to take formal specifications in Z, and associate each specification schema with the collection of code fragments that implement the specified operations. Our experience here was positive. The primary issue is the difficulty of writing queries to *automatically* identify program fragments that may be part

```
<Aggregate type="java_class" id="23">
  <Field name="name">
    <String>project.util.StringMap</String>
  </Field>
  <Field name="package">
  <Semantic type="Text" label="java_package_decl" id="27">
    package project.util;
  </Semantic></Field>
  <Field name="imports"><Set type="java_import_decl">
    <Semantic type="text" label="java_import_decl" id="42">
      import java.util.Map;
    </Semantic>
    <Semantic type="text" label="java_import_decl" id="43">
      import java.util.HashMap;
    </Semantic>
  </Set></Field>
  <Field name="decl">
    <Semantic type="Text" label="java_class_decl" id="29">
      class StringMap implements Map extends HashMap
  </Semantic></Field>
  <Field name="members"><List type="java_member">
    <Semantic type="text" type="java_member" id="48">
      public void putString(String key, String value) {
      ... }
    </Semantic>
    <Semantic type="text" type="java_member" id="49">
      public void getString(String key) { ... }
    </Semantic>
  </List></Field>
</Aggregate>
```

(a) Java Aggregate XML Form

```
package project.util;

import java.util.Map;
import java.util.HashMap;

/*+id=29*/
class StringMap implements Map extends HashMap {
  /*+id=48*/
  public void putString(String key, String value) {
  ... }
  /*+id=49*/
  public void getString(String key) { ... }
}
/*+AGGINFO
<Aggregate type="java_class" id="23">
  <Field name="name" type="String"
    value="project.util.StringMap"/>
  <Field name="package" type="java_package_decl">
    package project.util
  </Field>
  <Field name="imports" type="java_import_decl Set">
    <member id="42">import java.util.Map</member>
    <member id="43">import java.util.HashMap</member>
  </Field>
  <Field name="decl" type="java_class_decl" id="29"/>
  <Field name="members" type="java_member List">
    <memberref id="48"/>
    <memberref id="49"/>
  </Field>
</Aggregate> */
```

(b) Java Aggregate Java Source form

Figure 5: An Example of Stellation externalization

of a given operation implementation. We judged this a minor issue, because in specification-driven design, we know in advance exactly which program fragments should be associated with each specification fragment.

3.4.2 Aggregate Generation Performance

In order to evaluate the performance of aggregate generation, we selected a set of interesting queries used in the program understanding task from the previous section, operating over Jakarta Ant. We executed those queries over a Stellation repository containing the Jakarta Ant code using three different strategies:

1. **Baseline Strategy.** This technique knows the name of each repository artifact and uses knowledge of name encodings to reduce the candidate set. For each artifact in the candidate set, it then retrieves and analyzes the artifact to determine if there is a match. This algorithm is extremely inefficient, but is presented here to illustrate the impact of applying IR to this problem.

2. **Keyword Strategy.** This technique is the common vector-based information retrieval strategy, using an artifacts non-local identifiers as the indexed keyword set. It maintains an index of all non-local identifiers referenced within a fragment, and uses this keyword index to reduce the candidate set. For each item not disqualified using either name-encoding information or the keyword index; the associated artifact is then fetched and analyzed to determine if there is a match.

3. **Summary Strategy.** This is the full technique we describe. It maintains a non-local variable usage summary for each artifact.

For these tests, we generated three versions of the Stellation repository: one without summary information, one with keyword summaries of non-local variables, and one with complete usage summaries. Detailed information about generation time is not presented because the I/O time to store the code in the repository dwarfs the time needed to compute the summaries. In our tests, there was no measurable time difference for repository generation between the three versions.

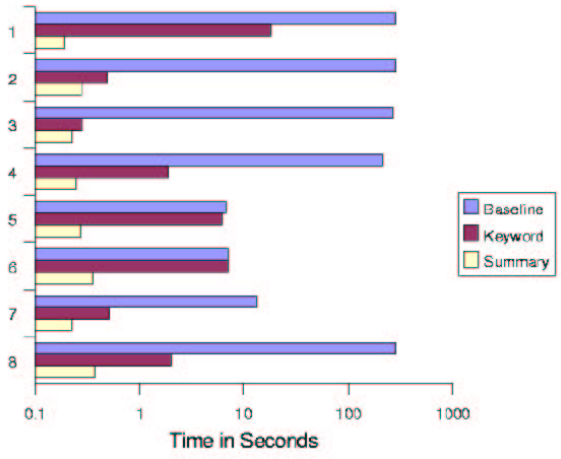
With this fast search system in place, if a developer can describe the desired aggregate in terms of the necessary queries, they can then produce the aggregate in a fraction of a second. With appropriate UI support, the process of describing a desired aggregate can be made quite simple and lightweight. The aggregate is created in a transient form, implemented as a workspace object without an assigned artifact ID. To convert the aggregate into a persistent repository artifact, it is only necessary to assign it a persistent ID, which can be done virtually instantaneously.

4. RELATED WORK

Aggregation constructs for coarse-grained SCM are nothing new: every major system in the last 20 years provides support for coarse grained aggregates in the form of directories or components. [**?**] provides a categorization of tools described in part by ther use of aggretion. However, our

```
1  all x | x assigns project
2  all x | x creates String AND
         x assigns excludes
3  all x | x assigns buildFile
         AND x assigns msgOutputLevel
4  all x | x creates BuildException
         AND x passes destDir
5  all x | x implements execute
         AND x creates BuildException
6  all x | x implements execute
7  all x | x implements execute AND
         x creates CommandLine AND
         (x invokes setValue OR
         x invokes createArgument)
8  all x | x creates CommandLine AND
         ((x invokes setValue) OR
              (x invokes createArgument))
```

(a) Queries over Jakarta Ant



(b) Query execution times

Figure 6: Performance measurements of queries over the Jakarta-Ant codebase

idea of using a flexible type-based aggregate system as an enabler for fine-grained SCM is novel. The merge annotation mechanism resembles some of the feature-logic merge specifications of Zeller's framework for describing merge behavior over coarse-grained aggregates[27, 28].

Fine-grained software configuration management has been explored by others, including the Gwydion/Sheets hypercode editor[25], which implements a query system much like ours, but in a program repository that does not support any kind of versioning; Desert[17, 23], which provides fine-grained SCM support, including a facility providing something very similar to our virtual source files; and Lemur[**?**] and COOP/Orm[1], which provide even finer-grained versioning than Stellation, but without significant aggregation support.

Our query algorithm is based on work done in the information retrieval (IR) community. A good survey containing an overview of the field can be found in [10]. Our approach is based on the vector summary algorithm used by Gerald Salton in the SMART system[24].

IR over program repositories has been done by several tools, such as eColabra/Asset Location[7], however these tend to focus on either asset management or software reuse. Other projects have done similar work using IR for program components, but instead of asset management, they have focused on identifying complete software components that are candidates for software reuse. For instance, Richard Helm and Yoelle Maarek[14, 18] did work using IR to identify reuse candidates from an object-oriented class library based on natural language queries.

Finally, IR techniques have been applied for fine-grained code reuse by systems like CodeFinder[12] and CodeBroker[26]. These systems use IR techniques such as latent semantic indexing to create an interactive program reuse environment. Like Stellation, CodeBroker is based solely on information extracted from the program. But CodeBroker generates its queries dynamically and automatically, refining them as the developer continues to type source code. Using these queries, CodeBroker selects artifacts in the repository using the cosine distance metric used by IR systems. All artifacts within a particular distance are presented to the user as potential reuse candidates.

Murphy and her students Baniassad and Robillard have explored methods for searching code repositories and identifying code that is part of a cross-cutting aspect. In [2], they describe a construct that encapsulates a cross-cutting concern in a software system, and in [22], they describe a method for examining code to identify the code that makes up a particular aspect.

Their *conceptual modules* are a construct that we believe can be modelled using Stellation's aggregate support. Their aspect discovery work closely resembles our usage summary technique (with finer granularity), and we believe that integrating usage summaries and Robillard concern graphs can provide an even stronger query mechanism for aggregate generation.

Program browsers, such as [3], Smalltalk[13], or more recently Eclipse[6], provide support for viewing, searching, and manipulating code in interesting ways. Eclipse and variants of Smalltalk[20] have even integrated these techniques with software configuration management; however, the capabilities provided by these systems do not approach those of the aggregate generation mechanisms that we have proposed.

The idea of multidimensional separation of concerns/spect-oriented software development has been explored in the software engineering community; an overview can be found in [**?**]. Most of the work in this field has focused on tools allowing developers to write systems using explicitly multi-dimensional semantic structures. These systems generally take one of two forms: tools that allow different perspectives and viewpoints (corresponding to different dimensions of concerns) to be reconciled [11, 9]; or systems that enable programs implemented with concerns separated using linguistic structure concepts to be integrated using program composition[19, 15].

Rather than providing another composition or reconciliation mechanism, our work has focused on the organizational aspect, keeping code in a single dimensional semantic structure, while allowing multidimensional organizational views. We believe our approach is complementary with the other approaches, and that multidimensionality is best supported through a combination of these techniques.

5. CONCLUSION AND FUTURE WORK

Fine-grained SCM offers substantial benefits for large-scale collaborative software development, particularly in the areas of complexity management and team communication and coordination. However, these benefits cannot be realized through fine granularity alone.

In this paper, we have proposed the comprehensive use of aggregates as first-order entities in order to manage the complexity explosion caused by large numbers of fine-grained artifacts. We claim that such reified aggregates also enable a host of other useful capabilities. By leveraging aggregation, an SCM system can support multidimensional program organization, improved collaboration, aspect discovery and separation, relationship management for heterogeneous artifacts, and more.

We argued that proper aggregation support for SCM systems has three key requirements:

1. a type system allowing developers to define both new kinds of aggregates and merge operations for aggregate artifacts;
2. a mechanism by which developers can quickly generate both transient and persistent aggregates;
3. effective round-trip interaction support for aggregates and other tooling.

In this paper, we presented the solutions provided by our SCM system, Stellation. These include a simple aggregate type system using annotations to define merge operations, a query system supporting rapid, flexible aggregate generation, a novel IR-based fast search mechanism, and an XSLT-based externalization/ internalization system for use with external and legacy tooling.

The performance and flexibility of the Stellation aggregate support was evaluated through an informal study of three moderate-size software projects and four task areas. Our overall assessment was positive, but we did find some limits in our ability to easily produce uniform models for in certain cases.

In future, we plan to add sophisticated user interface support making aggregates easier for developers to use, to explore new problems that we believe our approach can address, and to enhance our aggregate type system to provide better support for heterogeneous collections.

The Stellation system is an open-source software development project, and can be found on the web at "http://www.eclipse.org/stellation".

6. REFERENCES

[1] B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. In *ICSE '96 SCM-6 Workshop*, pages 31–48, 1996.

[2] E. Baniassad and G. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 64–73, 1998.

[3] Y. Chen and V. Rajilich. The c information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.

[4] M. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of FSE 2000*, 2000.

[5] M. C. Chu-Carroll. Supporting distributed collaboration through multidimensional software configuration management. In *Proceedings of the 10th ICSE Workshop on Software Configuration Management*, 2001.

[6] Eclipse platform technology overview. Technical report, OTI, Inc., July 2001.

[7] O. Edelstein, A. Yaeli, and G. Zodik. eColabra: An enterprise collaboration and reuse environment. In *4th International Workshop on Next Generation Information Technologies and Systems*, July 1999.

[8] James Clark (editor). XSL transformations (XSLT) version 1.0. W3c recommendation, W3C, November 1999.

[9] W. Emmerich, G. Spanoudakis, and A. Finkelstein. Next-Generation Viewpoint-based Environments. In *Proceedings of the 7th Workshop of the Next Generation of CASE Technology*, 1996.

[10] C. Faloutsos and D. W. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, University of Maryland, 1995.

[11] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedike. Viewpoints: a Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engienering*, 2(1):31–58, March 1992.

[12] G. Fischer, S. Henninger, and D. Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings 13th ICSE*, pages 318–328, 1991.

[13] A. Goldberg and D. Robson. *Smalltalk 80: the Programming Language*. Addison Wesley Longman, Inc., 1989.

[14] R. Helm and Y. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Proceedings of OOPSLA'91*, pages 47–61, 1991.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*, June 1997.

[16] D. LeBlang. The CM challenge: Configuration management that works. In *Configuration Management*, pages 1–37. John Wiley and Sons, 1994.

[17] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of ICSE 18*, pages 298–307, 1996.

[18] Y. Maarek, D. Berry, and G. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[19] H. Ossher and P. Tarr. Multi-dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology*. Kluwer, 2000.

[20] OTI. ENVY/Developer: The collaborative component development environment for IBM visualage and objectshare, inc. visualworks. Webpage: available online at: "http://www.oti.com/briefs/ed/edbrief5i.htm".

[21] Dave Pawson. An introduction to XSL formatting objects. Webpage at "http://www.dpawson.co.uk/xsl/sect3/bk/index.html", 2001.

[22] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependenceis. Technical Report UBC-CS-TR-2001-13, Department of Computer Science, University of British Columbia, 2001.

[23] S. Reiss. Simplifying data integration: the design of the Desert software development environment. In *Proceedings of ICSE 18*, pages 398–407, 1996.

[24] G. Salton. *The SMART Retrieval System - Experiments in Automatic Document Processing*. Prentice-Hall, Inc., 1971.

[25] R. Stockton and N. Kramer. The Sheets hypercode editor. Technical Report 0820, CMU Department of Computer Science, 1997.

[26] Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of FSE 2000*, 2000.

[27] A. Zeller. A unified configuration management model. Technical Report 95-03, Abteilung Softwaretechnologie, Technische Univesität Braunschweig, 1995.

[28] A. Zeller. Smooth operations with square operators: the version set model in ICE. In *ICSE '96 SCM-6 Workshop*, pages 8–30, 1996.