Connecting Architecture Reconstruction Frameworks

Ivan T. Bowman, Michael W. Godfrey, Richard C. Holt

University of Waterloo,
Waterloo, Ontario, Canada,
{itbowman, migod, holt}@plg.uwaterloo.ca

Abstract

A number of standalone tools are designed to help developers understand software systems. These tools operate at different levels of abstraction, from low level source code to software architectures. Although recent proposals have suggested how code-level frameworks can share information, little attention has been given to the problem of connecting software architecture level frameworks. In this paper, we describe the TAXForm exchange format for frameworks at the software architecture level. By defining mappings between TAXForm and formats that are used within existing frameworks, we show how TAXForm can be used as a "binding glue" to achieve interoperability between these frameworks without having to modify their internal structure.

Keywords: Architecture reconstruction, exchange format, program understanding, repository.

1. Introduction

Researchers have designed several standalone frameworks to help developers understand large software systems, including Ciao [4], Dali [10], ManSART [30], PBS [9], Rigi [19], SPOOL [15], and TkSee [16]. These frameworks are necessary because, for many systems, there is no high-level documentation that accurately describes the current system implementation. Even where documentation does exist, these tools are helpful because they can help developers assess how closely a system's implementation matches its documented structure.

Program understanding frameworks can help to answer questions about a software system at varying levels of abstraction. At the *code-level* of abstraction [14], frameworks can provide detailed answers to questions such as:

1. What expressions are affected by the value stored in variable *a* at a particular line in the program?
2. What are the possible values of variable *b* at a given line in the program?
3. How can this program be expressed in another programming language?
4. What functions may call function *f*?

Another important class of frameworks attempts to find the high-level abstractions that are used in software systems. These *architecture-level* frameworks (such as PBS and Dali) aid in reconstructing a system's software architecture based on facts extracted from the system artifacts—for example, source code, object files, Makefiles, and profiling results. Typically, these architecture-level frameworks are not designed to answer questions at the level of detail provided by the code-level frameworks. Instead, they address architectural questions such as the following:

1. What is a good subsystem decomposition of the system?
2. Does the implementation of a software system match its documented architecture? If not, where does it deviate?
3. What are the dependencies between subsystems?

The reconstructed software architecture provides a high-level description of the structure of a software system.

Recent work by Woods *et al.* [29] has suggested CORUM as a possible standard to connect frameworks that operate at the code level of abstraction. Kazman *et al*. [14] have extended the CORUM standard with CORUM II by incorporating concepts and tools from software architecture frameworks. However, the CORUM II proposal does not define an exchange data format for use between architecture-level tools. The lack of a standard exchange format does not prevent reuse between architecture-level frameworks, but it does make reuse difficult.

There are several reasons why it is important to be able to share data between architecture-level frameworks. First, it reduces the amount of duplicated effort that has been required to date. Second, it allows researchers to compare the results of applying their frameworks to the same input, which can help to identify strengths and weaknesses in the various tools. Finally, it allows developers who are attempting to reconstruct the architecture of a system to opportunistically use tools from each of the frameworks. For example, the developer could use a parser from one framework, a clustering tool from another, and a visualization tool from a third framework.

In addition to research-oriented frameworks, there are a number of commercial tools that provide support for reverse engineering and program-understanding activities, including Rational Rose [21], SNiFF+ [23], To-

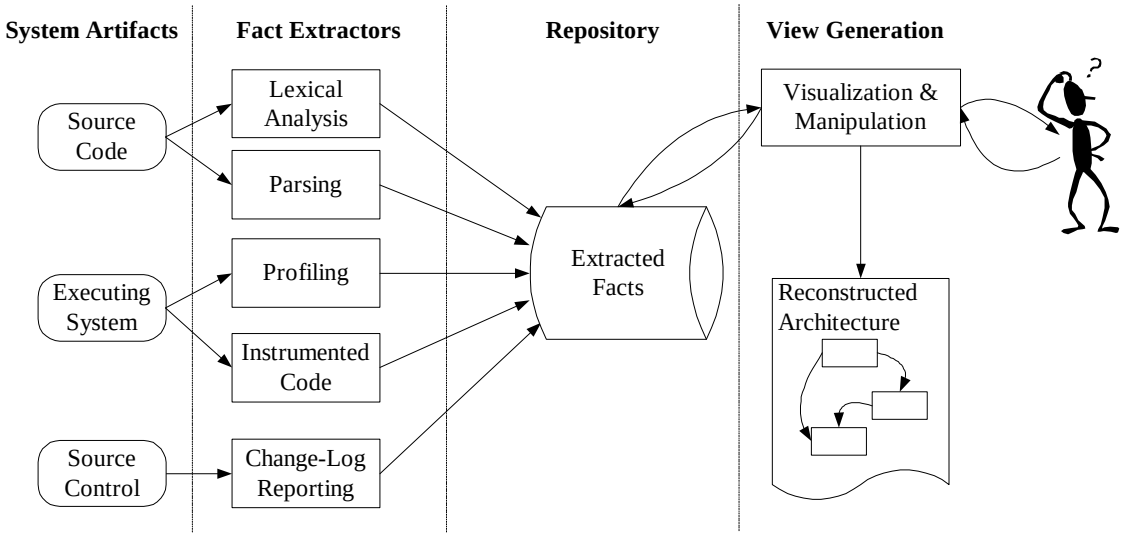

Figure 1: Overview of architecture reconstruction frameworks.

gether/Enterprise [26], and Visio [27]. Our work to date has focused on research-oriented tools for several reasons: researchers are motivated to re-use components between frameworks, and there are no restrictions on discussion of internal formats for non-commercial tools. Once we have designed a practical exchange format that can be used with research-oriented tools, we hope to investigate the creation of a standard that can also be used by commercial frameworks.

In this paper, we define an exchange format called *TA Exchange Format* (TAXForm). This format allows architecture-level frameworks to share extracted data. By defining a mapping between facts from each framework, TAXForm is intended to provide a common medium for exchanging information about software systems that is useful for reconstructing their architecture.

1.1. Organization

The remainder of this paper is organized as follows. Section 2 describes our motivation for connecting architecture reconstruction frameworks. Section 3 provides an overview of existing architecture reconstruction frameworks. Section 4 describes the requirements for a successful exchange format. Section 5 describes TAXForm and shows how it meets the requirements. Section 6 describes how we evaluated TAXForm by implementing several tools to convert from existing formats to TAXForm. Finally, Section 7 summarizes our results and describes future work.

2. Historical context

Several frameworks exist to reconstruct software architectures, and there are good reasons for this diversity. Researchers implement different research frameworks in order to experiment with novel techniques; industrial companies implement a variety of frameworks with differing features for competitive purposes. Storey *et al.* [24] found that different tools support different understanding strategies. The variety of existing frameworks allows developers to choose a program-understanding framework that is most suitable for technical, political, or budgetary reasons.

It is reasonable to expect that there will continue to be a variety of frameworks intended to help developers reconstruct the architecture of software systems. However, it is also desirable to be able to re-use components between these frameworks (where this is permitted by the framework owners).

Recently [8], researchers from five universities (Montreal, Ottawa, Toronto, Victoria, and Waterloo) and two industry groups (Bell Canada and the Software Engineering Institute) met to discuss the possibility of a common exchange format for information extracted from computer programs. Together, these researchers are responsible for the Dali [10], PBS [9], Rigi [19], SPOOL [15], and TkSee [16] tools, and all are interested in sharing results and components between tools. At this meeting, it was agreed that the creation of a common fact exchange format would be of significant mutual benefit. This paper describes our work resulting from these discussions.

3. Overview of architecture reconstruction

While there are several different approaches to reconstructing the architecture of a software system, there are also several common requirements. An architecture recovery system extracts *facts* from a system implementation, then combines these facts into higher-level abstractions. The extracted facts may be of many forms. Researchers have extracted information about function calls, data accesses, file operations, and network communications to help reconstruct views of a system's software architecture. Furthermore, individual systems may have

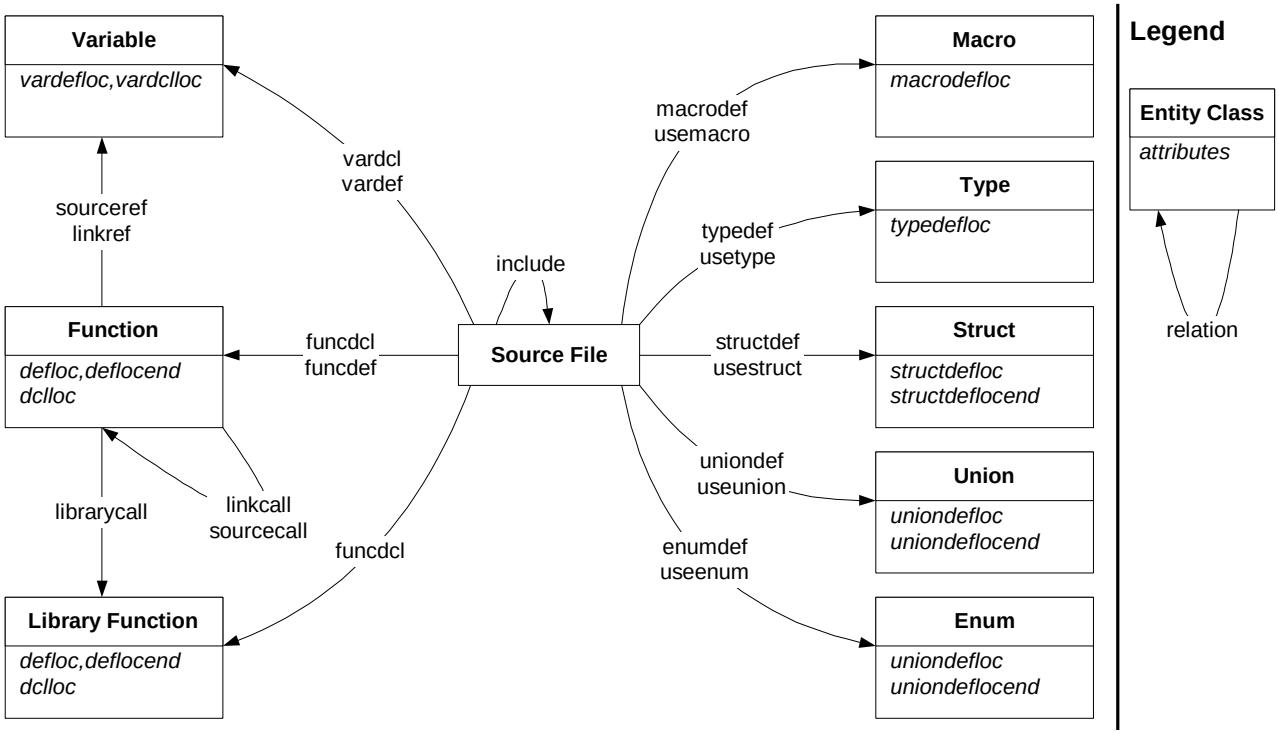

Figure 2: PBS Low-level schema for the C programming language.

system-specific facts that can be used to help understand the system's structure.

Figure 1 shows an overview of architecture reconstruction frameworks (such as PBS [9] and Dali [10]). These reconstruction frameworks use *fact extractors* to determine facts about a system implementation. These frameworks extract facts from system source code, executing instances of the system, and source control logs. All of these extracted facts are stored in a repository. Next, a developer interacts with an abstraction tool (which is often visual) to combine elements into higher-level abstractions that describe the system architecture.

PBS, Rigi, and Dali use an *entity-relation* (ER) model [3] to store facts extracted from the system implementation. Code elements (such as functions, variables, data types, and source files) are represented as *entities*, and the relationships between these entities (such as function calls, uses of variables, and instantiations of data types) are represented as *relations*. The possible entities and relations are defined in a *schema*. Each entity may also have associated *attributes* that describe properties such as the entity's name or line number.

In addition to the facts extracted from the system implementation, these frameworks also store information created by the framework and its users. For example, this information might include a clustering of entities into groups, as well as information pertaining to a visualization of the structure. TAXForm does not yet have support for these facts such as visualization information; this will be addressed in future work.

Each of PBS, Rigi, and Dali use distinct schemas to represent the facts extracted from system implementations and the derived facts created by the framework. Several schemas may be used within a framework depending on the particular source-code language used in the system and the source of the extracted facts. Kazman and Carrière [12] have described how it is useful to combine facts from several different sources to complement weaknesses in each particular fact extraction. For example, they augmented static analysis of function calls with profiling information to determine which instance of a virtual function was invoked. Experience with these frameworks shows that it is often necessary to create new schemas that can be easily used with the framework.

3.1. The PBS source models

In this section, we describe the schemas used by PBS as an example of the schemas used in architecture reconstruction frameworks. When working with programs written in the C programming language, PBS uses a fact extractor called *cfx* to extract a source model of the system. This source model is used to derive a higher-level model that shows relationships between source files. Finally, a developer interacts with a visual abstraction tool to group source files into related subsystems. The PBS system then calculates relations between these newly created subsystems based on the relations between the contained files.

Figure 2 shows the schema used by PBS to store facts extracted from systems written in C. This schema contains entity classes, relation classes, and possible attrib-

utes. The schema defines what types of entities are allowed, the relations that are permitted between these types of entities, and the attributes that may be stored for each entity or relation. For example, the schema described in Figure 2 allows us to store information about two source files (*A* and *B*), and a *include* relation between *A* and *B*. However, this schema does not allow us to store a *librarycall* relation between *A* and *B*, since that type of relation is not permitted between entities of type *Source File*. The PBS source model is described in more detail in the PBS documentation [9].

The schema shown in Figure 2 represents a detailed source model of C programs. The analysis PBS performs doesn't require this level of detail. In fact, the detailed facts make it difficult to visualize and understand the extracted information. Instead, PBS abstracts facts stored in the source model into a high-level model of the software system. PBS uses the Grok [10] tool to perform this abstraction. In addition to the facts that are derived from the system implementation, PBS uses a subsystem containment hierarchy that is defined by a developer to represent groups of related files.

Figure 3 shows the high-level schema that is used by PBS. This schema models facts between source files (represented by the *Module* entity class) and *Subsystems*. Both *Module* and *Subsystem* entity classes inherit from *Software Element* (this means that they inherit all of the possible attributes and relations that can appear for *Software Element*). For example, we can record a *usevar* relation between a subsystem and a module. The *contain* relation is used to record the subsystem hierarchy.

The PBS high-level schema is quite abstract, and this high level of abstraction helps developers understand large systems by reducing the clutter caused by low-level details. PBS has been used successfully with this schema to reconstruct the software architecture of several large systems, including the Linux [2] operating system (containing more than 800 KLOC).

4. Requirements for a connection standard

For architecture-level frameworks to share data, they must all conform to a connection standard of some kind. This standard could define an API that frameworks use to read and write facts (this approach is used in the CORUM standard). Alternatively, a connection standard can define a file format that each framework must be able to read and write. We have chosen the latter approach in defining TAXForm. A connection standard defines the syntactic form of a file format, and the schema of the facts stored in the file format.

For a connection standard to be successful, it must satisfy several criteria. Müller [18] proposed the following criteria:

1. It should work for several levels of abstraction (*e.g.*, code-level and architecture-level).
2. It should work for several source languages (*e.g.*, C, C++, Java, PL/I, Cobol, RPG).
3. It should scale to large systems (*e.g.*, 3 to 10 MLOC).
4. It should support a mapping between entities/relations and the source code statements.
5. It should work for static and dynamic dependencies.
6. It should be incremental, so that it is possible to add one subsystem at a time.
7. It should be a universal standard that is widely accepted.
8. It should be extensible, allowing users to define new schemas for the facts stored in the format as needed.

Since we are only concerned with exchanging information at the software architecture level, we do not consider criteria 1 in this paper.

In addition to these requirements, we have identified three particular problem areas that must be addressed by a portable exchange format.

4.1. The naming problem

In an entity-relational model, each entity must have a unique identifier. If a fact extractor has simultaneous access to all of the source-code entities, then a unique number can be assigned to each entity and used as an identifier. This approach, however, makes it difficult to combine the results of different fact extractors since they will assign different unique numbers to the entities they extract. In addition, if entities are identified by number then it is difficult to compare two versions of a software system since it is likely that corresponding entities will be assigned different numbers in the two versions.

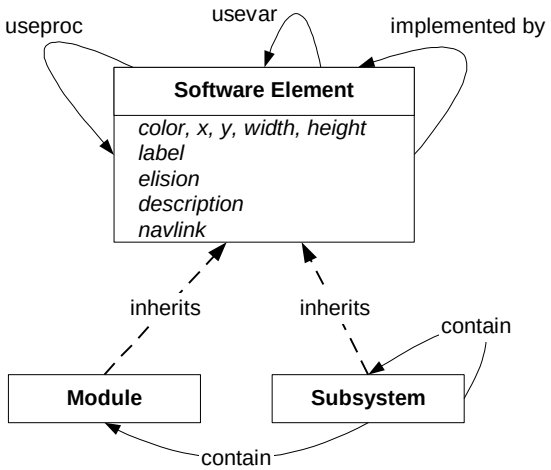

Figure 3: PBS high-level schema.

Instead of using numbers, existing architecture reconstruction frameworks (such as PBS and Rigi) create an identifier for each entity using the name of the entity in the source code. This approach is also not perfect because several languages allow a single name to be used for different entities. For example, the C language allows a variable and a label to have the same name. Languages such as Java and C++ allow function names to be over-

loaded based on the data types of their parameters. Most modern languages allow identifiers to be re-used provided they are in different scopes.

If architecture reconstruction frameworks are to combine extracted facts from multiple extractors, they must assign consistent names to source code entities. Consistent names can be created using *mangled names*. A mangled version of a source-code identifier would contain the *name-class* of the entity (for example, label or variable), and any required scoping information. For languages such as Java and C++, the mangled name would include the data types of the parameters of functions in order to distinguish between overloaded instances of a function.

For languages such as C, C++, and Java, there is a mangling algorithm that is used by compilers and linkers for identifiers that have *external linkage*, that is, global variables and functions. These language-defined mangling algorithms do not typically apply to local variables or data types.

A portable exchange format should define a standard way to uniquely identify source code entities. This mechanism should be based on existing mangling mechanisms defined by the source language, with extensions to handle entities that are not uniquely identified by the languages (such as local variables and data types).

4.2. The resolution problem

Fact extractors detect when one source code entity refers to another, and record this as a relation. The algorithm that the extractor uses to determine which entity is referred to depends on the source language and the implementation of the extractor. There are four categories of resolution produced by fact extractors.

1. **Not resolved.** Lightweight extractors that use only lexical information such as regular expressions do not store enough context to resolve the use of an identifier to any previous declaration. Instead, these extractors use the name of the code entity as the entity identifier. This approach makes it easy to write extractors. However, several source code entities with the same name may be indistinguishable in the extracted facts.
2. **Resolved to declaration.** Typically, compilers resolve each use of an identifier to a declaration of the identifier. If the identifier has not been previously declared, it is implicitly declared by its use in the source code. A code entity may have several declarations, but only one definition.
3. **Resolved to static definition.** When a software system is compiled and linked, each source-code entity has a single definition. The linker resolves all of the references to global variables and functions to the appropriate definition. There is no corresponding data type linker for most compiler implementations.
4. **Resolved to dynamic definition.** Languages that support dynamic binding (such as C's function pointers and C++ and Java's virtual functions) can have relations between source code entities that can not be statically determined. For these relations, we must look at the dynamic behavior of the software system to determine which source code entity is selected for a given identifier at run-time. Extractors such as profilers can determine which source code entity is chosen in a particular run of a software system.

Different extractors emit facts with these four levels of resolved references. Some extractors have goals that prohibit them from accurately emitting references resolved to category 3 or 4. For example, Lethbridge and Anquetil [16] describe how conditional compilation prevents an extractor from determining a unique target for a reference—there may be several possible definitions of a given symbol, each one selected based on different build-time options. Tools such as *cfx* (used in PBS) extract facts only for a single system configuration. Since these tools examine a particular configuration, they can choose a unique binding for a reference—the same binding that is chosen by a compiler building the system with the same configuration options. However, this restricts the extracted facts to a single system configuration instead of all possible configurations. In different configurations, a reference may bind to a different target, or not be bound to a target at all.

To determine dependencies between systems, we need extractors that support category 3 or 4. If an extractor produces facts resolved to category 1 or 2, we can use it as if it resolves references to category 3, or even 4. To do this, we assume that each use of an identifier refers to all entities with the same identifier. This approach produces an overestimate of the relations within a system. This overestimate of relations can be useful in some circumstances; for example, we can compare two fact extractors (A and B), where A produces facts resolved to category 3 and B produces facts resolved to category 1. We can treat facts extracted from B as if they were resolved to category 3 if we make the overestimate assumption. If A extracts relations that are not extracted by B (despite the overestimate assumption) then we can determine that either A is incorrectly producing relations that do not occur in the system implementation, or B is missing relations that may occur. If on the contrary, B extracts relations that are not extracted by A, we can determine nothing in general because of our overestimate.

4.3. The line number problem

One attribute that can be recorded for source-code entities is the location where they are defined within the system's source code. Conceptually, this consists of recording the path to the source file that contains the entity, and the line number that contains the entity's definition.

In practice, there are some complications. First, it is desirable that the location be reproducible if the extraction is performed on a different machine. In this case, the path to the system's source code may be different. For reproducibility, we should store all file names relative to the root of the system's source code. Second, some source-code entities are not defined on a single line (or even in a contiguous range of lines). For example, a C++

namespace entity can occupy several distinct regions of multiple source files. In addition, the pre-processor used for C and C++ allows source code entities to span source files. If we wish to accurately store the definition locations of all entities, we must store a set of location ranges. Each range would store the path name of a source file relative to the root of the software system, and the beginning and ending character of the range.

The level of detail for storing entity locations varies between extractors. Some extractors store no information, others store the file name, beginning and ending line number. Others store a sequence of file name, beginning character and ending character. A portable exchange format should support these possibilities and provide a mechanism for extractors to indicate what form of entity location they are providing.

5. TA Exchange Format (TAXForm)

Figure 4 shows an example of how architecture level frameworks can be combined to help reconstruct the architecture of a software system. In this example, a developer uses extractors from the Dali, Rigi, and PBS frameworks to extract facts from the system's source code. As described by Armstrong and Trudeau [1], each of these extractors has individual merits that might lead a developer to combine their results to get an accurate model of the source code. After combining the extracted source model into a repository, the developer uses the Bunch clustering tool [17] in combination with the PBS viewer and abstraction tools to find a good subsystem decomposition of the system being considered. These tools store the derived subsystem decomposition in the TAXForm repository. Finally, the developer prepares an animated presentation of the system's structure using the SHriMP viewer tool [24] that is part of the Rigi framework. In future work, we will extend TAXForm so that it can store the views generated by the PBS viewer and Rigi's SHriMP tool. By using converters to change between a framework's internal format and TAXForm, we allow existing frameworks and tools to work with each other without changing their internal structure.

To enable the example described in Figure 4, we need to define the TAXForm repository format and then implement the converters that are used in the example. To define TAXForm, we must define both the syntax of the repository format and the schema of the facts that are stored within the files. The schema defines what entities, relations, and attributes are stored in a TAXForm repository. TAXForm defines schemas for different levels of abstraction and source language. For example, TAXForm defines a high-level 'module dependency' schema that records a single relation (depends-on) between modules. TAXForm also defines a more detailed schema for procedural languages (TAXForm schemas are discussed in more detail in Section 5.3).

TAXForm defines several schemas for facts extracted from software systems. However, as discussed previously, developers often need to create new schemas to describe entities and relations that are unique to a particular system or problem domain. TAXForm must be able to accommodate these user-defined schemas in such a way that facts stored in these schemas can be used by visualization and abstraction tools that read and manipulate the extracted source model. TAXForm achieves this extensibility by allowing users to define a transformation between their user-defined schema and a TAXForm schema. In this way, an abstraction tool can operate with a user-defined schema by first transforming the facts into the appropriate TAXForm schema.

The remainder of this section is organized as follows. Section 4.1 describes the syntax of TAXForm. Section 4.2 describes how we can transform facts from one schema to another. Section 4.3 describes the schemas that are defined by TAXForm. Finally, Section 5.4 describes the overall structure of a TAXForm repository.

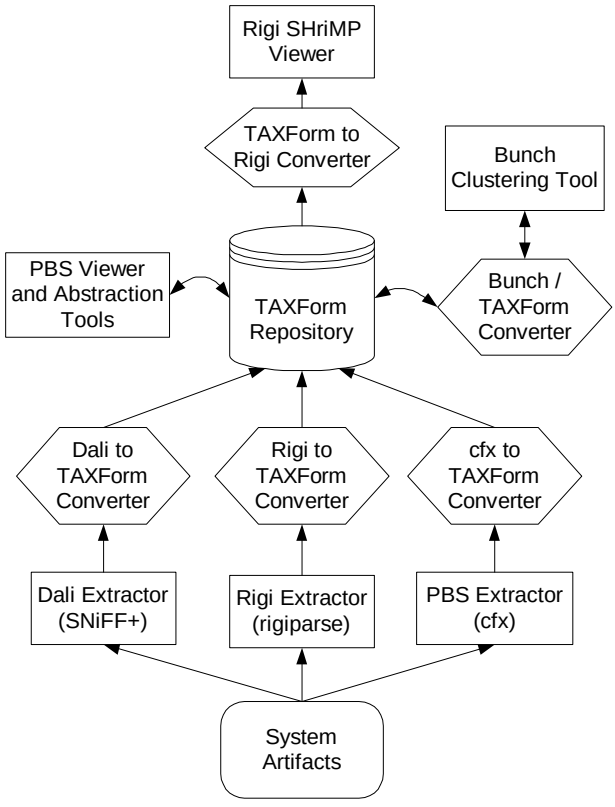

Figure 4: Example of connecting frameworks.

5.1. Exchange syntax

The syntactic form of the exchange format is an important point of standardization. Simple formats that are easy to produce and consume have traditionally proved quite useful. To date, several syntactic forms have been used to store facts about software systems: Rigi uses the

Rigi Standard Format (RSF) to store facts as tuples [28]. PBS uses the Tuple-Attribute (TA) language [7]. Dali and CIA use a relational database.

Other syntactic forms are appealing. For example, XML is a standard format for describing and exchanging structured data. Designing a useful exchange syntax is an interesting problem for future study. For now, we propose using TA as an exchange format. TA is a simple text-based format that is easy to read and write.

A TA file consists of four sections:

- The **Scheme Tuple** section describes the schema of possible entity classes and relations between these classes.
- The **Scheme Attribute** section describes all of the possible attributes for each entity class and relation class.
- The **Fact Tuple** section records entities and relations. Each entity is defined using the $INSTANCE relation, which defines an entity's class.
- The **Fact Attribute** section defines the attributes for each entity and relation.

```
// A comment line
SCHEME TUPLE :
call function function

SCHEME ATTRIBUTE :
function {
  static_fn
}
(call) {
 line_number
}

FACT TUPLE :
$INSTANCE f1 function
$INSTANCE f2 function
call f1.c f2.h

FACT ATTRIBUTE :
f1 {
  static_fn=true
}
f2 {
  static_fn=false
}
(call f1 f2) {
  line_number=2
}
```

Figure 5: An example of TA

Figure 5 shows an example TA file that records information about function calls in C source code. When extracting facts from a system, there would be one such TA file for each source file in the system; these files could later be combined into a single TA file representing the entire system. The TA file describes the schema (allowed entity classes, relation classes, and attributes) and facts (actual entities, relations, and attributes). In this example, the schema has one entity class (*function)* and one relation (*call*) that can connect functions. The TA file also specifies the attributes that can be stored. The *function* entities have an attribute (*static_fn*) that is true if a function is static instead of global. The *call* relation can have an attribute (*line_number*) that specifies the line number where the function call occurs. In addition to the schema, the TA file contains extracted facts. This example has two functions (*f1* and *f2*). There is a call recorded from *f1* to *f2*. Finally, the TA file records the attributes of the entities and relations. In this example, *f1* is a static function, while *f2* is a global function. The call from *f1* to *f2* occurs on line 2 of the source file associated with the TA file.

The PBS documentation [7] defines the TA syntax in detail. We are continuing to work on defining extensions to the TA format to support exchanging facts. For example, TA files can contain comments that name the developer that extracted the facts and the tools that were used during the extraction. We are defining a standard form for these comments to make it easy for tools to automatically extract this information.

5.2. Transforming facts to a different schema

TAXForm allows developers to define new schemas; these schemas are integrated into the TAXForm repository by transforming them as necessary into a form that tools can use. For example, facts extracted by PBS can be converted from the PBS C language schema to the TAXForm C schema.

If we wish to transform facts from one schema (S1) to another (S2), then we must define an algorithm that defines how entities, relations, and attributes will be created in the target schema based on the extracted facts. Holt describes how these types of transformations can be performed using Tarski relational algebra [10]. The transformation can be encoded in an algorithm in the Grok language.

One case where transforming the facts is trivial occurs when the source schema (S1) is a subset of the target schema (S2). If S2 contains all of the entity classes, relations, and attributes that are found in S1, then facts that conform to S1 also conform to S2. In this case, no transformation is needed.

In general, S1 will not be a subset of S2. For example, S1 may represent facts at a lower level of detail (functions and variables) than S2 (file level facts). In this case, the transformation algorithm must combine all of the function and variable entities in S1 into the files that contain them. Then, the algorithm must induce relations between files based on the relations between the contained functions and variables.

In addition to transformations based on entities and relations, it may be necessary to perform transformations that consider the values of attributes for entities and relations. For example, S1 might store all functions (static or global) as a single entity class, using an attribute to distinguish between the two types of function. If S2 uses different entity classes for global and static functions, then the transformation algorithm must choose the target entity class based on the entity class in S1 and the value of the distinguishing attribute.

Another type of transformation that may be needed is a renaming of entity identifiers. As discussed in Section 4.1, different fact extractors may choose different methods to uniquely identify extracted entities. Transforming facts from one schema to another may require that the naming scheme be changed to match.

A variety of rules can be used to define transformation algorithms from facts stored in one schema into facts that conform to another schema. However, it is important to ensure that these transformations preserve the meaning of the stored facts. For example, it probably does not make sense to transform a function in one schema into a variable in another schema. It is possible to define such a transformation, but the result is probably not useful. When defining transformation algorithms between schemas, developers must consider the meaning of the entities and relations.

5.3. TAXForm schemas

The facts that are used when reconstructing software architectures come from a variety of programming languages using a variety of fact extractors. The wide variance in semantics of various languages and extractors makes it difficult (if not impossible) to define a single schema that can accurately represent all of the information that various frameworks wish to store. Instead of trying to define a single such schema, TAXForm defines a set of schemas and describes transformation algorithms between them.

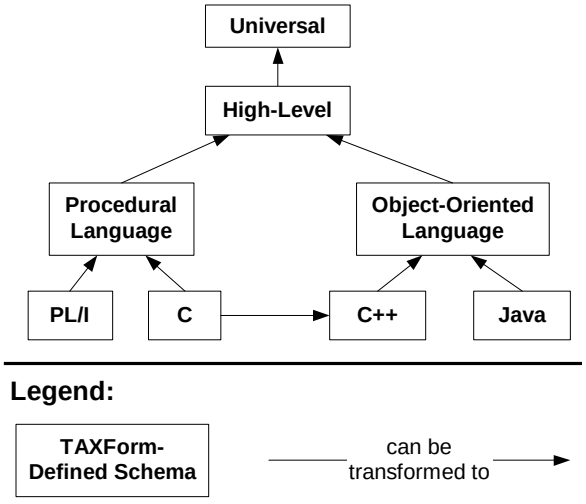

Figure 6: Transformations between schemas.

Figure 6 shows a subset of the TAXForm-defined schemas and how they can be transformed to each other. These schemas vary in their level of abstraction (with the Universal schema being most abstract), and the domain of languages to which they apply.

Figure 6 shows arrows between two schemas if there is a transformation algorithm from one to the other. In general, low-level schemas can only be transformed meaningfully into higher-level schemas. An exception is that the schema for the C language can be transformed into the schema for the C++ language since C is a subset of C++.

When combining facts from two different extractors, we can transform both sets of facts into a common TAXForm-defined schema. If both extractors record the same type of information (such as function calls), then we can find a *least-common-denominator* schema, that is, a well-defined schema that can describe the facts extracted by both extractors. If they extract different types of information (for example, one extractor shows function calls, the other shows which developers worked on a file), then we can create a new schema that is a composite of both extraction schemas. This composite schema can be defined as a TAXForm schema in terms of the definitions of its constituent schemas, including the associated transformations.

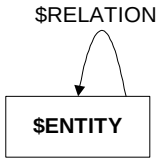

Figure 7: TAXForm Universal Schema.

Figure 7 shows the most-abstract schema possible—the Universal Schema. This schema contains only a single entity class and a single relation type (the names are taken from TA's documentation [7]). Any particular schema can be transformed into the Universal Schema by mapping all entity classes to $ENTITY and all relation types to be $RELATION. The existence of a most-abstract schema guarantees that facts from two sources can always be combined into a common fact base, although the results may be too abstract to be useful.

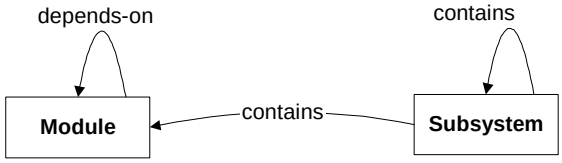

Figure 8: TAXForm high-level schema.

To ensure that extracted facts are sufficiently concrete to be useful, we must define a set of schemas that are less abstract. Figure 8 shows a high-level schema that describes software systems at an abstract level. This schema contains two entity classes: modules, which correspond to architecturally relevant groups of resources (a source file might be stored as a module), and subsystems. Subsystems contain modules as well as other subsystems—this containment hierarchy is usually defined as part of the reconstruction of the software architecture. This schema stores a single extracted relation, *depends-on*. A module may depend on another module for several reasons: it might call a function or access a variable defined in the

other module, or it might open a socket connection that is accepted by the other module. No distinction is made in this schema between the many forms of dependency.

This high-level schema is still quite abstract, but it can be used for several interesting forms of analysis. For example, we can determine if an as-built architecture deviates from the intended architecture by finding a dependency between modules in subsystems that are supposed to be independent. We can identify subsystems that can be reused in other systems, and determine what other systems may be required based on the dependency information. If an architectural abstraction tool (such as the Bunch clustering tool [17]) uses only the schema in Figure 8, then it can work with all of the TAXForm-defined schemas since they can all be transformed to this schema.

The high-level schema of Figure 8 can be elaborated for specific programming language types. Figure 9 shows a schema that represents a source model of procedural languages such as C or PL/I. This schema records information about source files, data types (*e.g.*, structures, unions, and enumerations), procedures, and data (*e.g.*, variables, enumerators, and manifest constants). The schema indicates where entities are defined: either at the global scope of a source file, or within a procedure. The schema also records *uses* information between the entities. A source file may use another file, for example by the C `#include` pre-processor directive. A procedure may use data types (for example, as a return type, parameter type, or cast type), data entities (for example, reading or writing a variable), and other procedure entities (for example, for function calls, or to take the address of a procedure). Data entities use their data type. The schema described in Figure 9 is more detailed than the high-level schema, and it supports more precise forms of architectural analysis. For example, the as-designed architecture for a system may stipulate that one subsystem contains procedures that may be called from outside the subsystem, but that no data should be accessed from outside the subsystem. Using the schema in Figure 9, we can evaluate whether the as-built architecture conforms to this stipulation.

The schema in Figure 9 could be elaborated still further to model details that are unique to a particular programming language, such as C or PL/I. Also, TAXForm can define schemas for object-oriented languages such as Java and C++. We are working to define these schemas based on existing formats used in architecture reconstruction frameworks.

5.4. The structure of a TAXForm repository

A TAXForm repository must contain (or refer to) several documents and facts, including the following:

1. A schema describing the entities, relations, and attributes that are stored in the repository.
2. Algorithms to transform between the schema being used and other TAXForm schemas.
3. A document describing the meaning of the entities, relations, and attributes.
4. The facts extracted (or derived) from the system implementation.

Only the schema and extracted facts are stored explicitly in the TA files containing the repository. The transformation algorithms and documentation are identified by a reference in a comment in the TAXForm file. For example, a TAXForm repository might refer to the PBS framework documentation if it conforms to PBS's documented behavior.

The transformation algorithms can be formalized using Tarski relational algebra, and executed using the Grok tool [10]. These algorithms need only be defined once for each schema, and referred to by the repositories that use the schema.

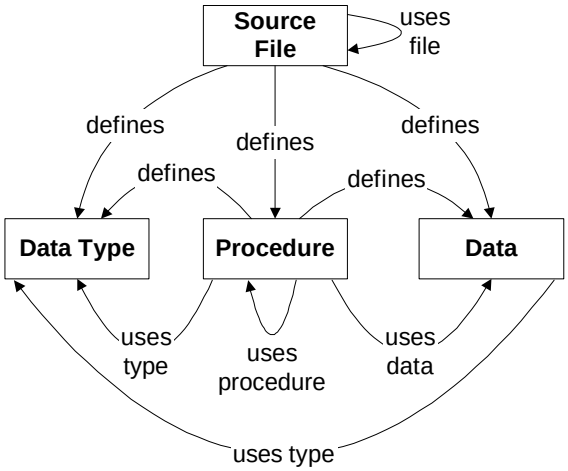

Figure 9: TAXForm procedural language schema.

Schema documentation requires that we document the meaning of entities, relations, and attributes. Developers will use this information to determine how to create transformation algorithms without making logically inconsistent mappings (such as mapping functions to variables). This documentation might be written in a developer's natural language, or expressed more formally. For example, Kazman *et al.* [13] suggest one set of features that can be used to characterize architectural elements. In addition to documenting the meaning of the contained facts, this documentation must identify how the schema solves the naming problem (see Section 4.1), the resolution problem (see Section 4.2), and the line-number problem (see Section 4.3).

Thus, a TAXForm repository consists of a set of files in the TA format. These files specify a schema (in the TA schema sections), and extracted or derived facts (in the TA fact sections). These files also refer to documentation and transformation algorithms that allow the repository to be converted to another schema.

Developers that use a TAXForm repository do so by choosing (or defining) a schema that satisfies the analysis they wish to perform. Next, facts that do not conform to this schema are transformed into the schema using the

defined algorithms. The developer uses tools that operate on the facts as though they were defined in the desired format. If no transformation to the desired schema is defined, then the developer can consult the documentation associated with the repository to determine if such a transformation is meaningful. If so, the developer can define a new transformation algorithm based on the documentation. Otherwise, it is not meaningful to apply the tool to the TAXForm repository.

6. Evaluation of TAXForm

In order to evaluate the effectiveness of the TAXForm exchange format, we implemented conversion utilities that translate from tool-specific notations into TAXForm. To date, we have implemented converters for Ciao [4], Dali [10], Datrix [5], PBS [9], and Rigi [19]. We have also designed a converter for TkSee [16] based on its documentation. By implementing tools that convert from the formats used by these frameworks, we were able to gain insight into the suitability of TAXForm as a general exchange format.

The first operation we performed in implementing our converters was to convert the facts into the TA syntax, while keeping the facts in the tool's original schema. For Ciao and Dali[1], this involved executing queries against a relational database to emit a text file containing both the schema definition and the extracted facts. The Datrix tool emits an AST; we implemented a tool that reads this AST, and emits a TA file that contains facts in an entity/relational model. The Rigi tool emits facts in Rigi Standard Form (RSF), which is easily converted to TA format by combining it with a schema definition. Finally, the PBS, and TkSee tools are already using the TA file format, and thus did not need converters to match the TAXForm syntax. In general, we found that it was easy to convert from tool-specific storage formats into the TAXForm syntax, provided we have access to documentation describing the tool-specific format.

After converting to the syntactic format of TAXForm, we attempted to define mappings from the different schemas into the TAXForm procedural language schema described in Figure 9. One issue that was difficult to address was the consideration of resolving references. Dali, Ciao, and PBS use a 'linker' that resolves all references to the target definition. In contrast, Datrix, Rigi, and TkSee do not use such a linker. For these three frameworks, some references identify the declaration of the target, not the definition. The Datrix team is developing such a linker. For Rigi, we defined a linker based on the semantics of the C language. For TkSee, the designers allow developers to examine partial or inconsistent systems; for these systems, there is no way to correctly resolve references. If we restrict ourselves to complete and consistent systems we could implement a linker for the facts extracted by TkSee.

[1] The Dali translator was implemented by Jeromy Carrière.

We were able to define simple transformations that convert from the schemas used by Dali, Ciao, and PBS into the TAXForm procedural language schema, and we implemented these transformations with the Grok [10] relational calculator. For Rigi, we implemented a linker using Grok, then converted to the TAXForm procedural schema using transformations defined in Grok. For Datrix and TkSee, we designed transformations that can convert the extracted facts to the TAXForm schema, but we have not yet implemented these transformations.

Overall, we found that it was relatively easy to convert to the syntactic form of TAXForm. However, we found that transforming from tool-specific schemas to the TAXForm schema required careful study of the tool-specific schema; each tool uses different terminology with different underlying assumptions about the semantics of stored facts. We found that we need to consider the issues raised in Section 4 when implementing extractors: the frameworks we examined used different naming conventions to identify entities, associate line number information, and resolve references.

In order to validate our translators, we exercised them with source models extracted from several real-world systems. Table 1 summarizes the subject systems that we used to test our translators.

System	Size	Language	Tool
Jikes	77 KLOC	C++	Ciao
Linux	800 KLOC	C	Dali, PBS
Mozilla	904 KLOC	C	Rigi
Nachos	10 KLOC	C++	Ciao

Table 1: Summary of systems used with translators.

We used Ciao to extract a source model from both Nachos, an operating system simulator used for education, and Jikes, a Java compiler. Both of these systems were written in C++. We found that we were able to convert from the Ciao format to TAXForm by writing queries that operate on the Ciao database. Ciao stores file and line numbers for each entity, and identifies all entities using unique integers.

We also examined Linux, a Unix-like operating system. We used two source models for Linux, extracted by PBS and Dali. The PBS tool creates a source model that is in the TA format, but not in the TAXForm procedural language schema. We used transformations written in Grok to transform to the TAXForm schema. PBS uses the source-code name of entities as their identifiers; this means that PBS cannot store facts about two different entities with the same source code name (for example, two static functions in different files). PBS also stores the source file name for all entities. The Dali group provided us with a source model of Linux in TA format, along with documentation describing the semantics of how they extracted this model. We were able to transform this model into the TAXForm procedural language schema. The facts extracted by Dali used the enclosing source file name as part of the unique identifier for entities (allowing

for multiple entities with the same simple name), but did not record line number information.

Finally, we converted a source model of the Mozilla web-browser that was extracted by the Rigi group. This source model represents facts about 904KLOC of C code (it does not model the C++ portion of Mozilla, which is about 800KLOC). We converted from the Rigi format to the TAXForm schema by first implementing a linker, then transforming to the new schema. The Rigi source model uses entity identifiers that are composed of the source file name, source code name, and line number. This structure allows Rigi to represent facts about different entities with the same simple name. We were able to translate the file name and line number information as extracted by Rigi into the TAXForm format.

We were able to use the converted facts for each of these large, real-world systems with PBS. Each of the frameworks we considered had different formats for entity identifiers and source code locations. We were able to convert all source code locations to a single format, but we have not yet implemented a solution that converts all entity identifiers to a common format. This means that we can not yet combine source models extracted by different frameworks, although we can use each of these source models independently. We are continuing to work on the issue of entity identifiers. It appears that we need to modify the existing frameworks to provide more information to permit us to do this mapping.

7. Conclusions

To date, no significant reuse has been possible between frameworks operating at the software architecture level of program comprehension. Instead, developers of these tools have re-implemented functionality such as parsers, clustering tools, and visualization tools. Reuse has been problematic because there is no standard format for exchanging facts between these frameworks.

In this paper, we have described the TAXForm exchange format for representing information that is useful to frameworks performing architectural reconstruction. Instead of defining a single schema, TAXForm defines a hierarchy of related schemas and shows how extracted facts can be transformed between various schemas. Each extracted fact base records the semantics of the schema it conforms to, and also records how this schema can be transformed to other schemas. This mapping information can be used to combine facts from two different extractors into a common schema. Since TAXForm does not define a single authoritative schema, developers are free to create a schema that best fits the facts that are interesting for a particular system (provided that they describe how their schema can be transformed into other schemas they wish to operate with). This flexibility, in combination with the simple text-based syntax of TAXForm, suggests its use as a standard exchange format for frameworks operating at the architecture-level.

We have implemented TAXForm converters to work with facts extracted by Ciao, Dali, PBS, and Rigi. By using these converters, the PBS framework is able to visualize and manipulate facts extracted by these frameworks; if other frameworks are modified to read the TAXForm exchange format, they will also be able to operate with these facts.

By providing a standard format for exchanging information, TAXForm allows researchers to compare experimental results and makes it possible for developers to opportunistically choose tools from different frameworks to best solve the problem of reconstructing a system's architecture.

Acknowledgements

We would like to thank several people who have helped us to develop the ideas presented in this paper, during meetings at WCRE'98 and CASCON'98. Nicholas Anquetil, Jeromy Carrière, Gary Farmaner, Ruedi Keller, Bruno Lague, Tim Lethbridge, Hausi Müller, Patrick Page, Daniel Proulx, Derek Rayside, Reinhard Schauer, and Kenny Wong contributed at these meetings. In particular, we would like to thank Tim Lethbridge for clarifying the issues related to resolving references. In addition to providing helpful feedback, Jeromy Carrière implemented the translator from Datrix to TAXForm. We would like to thank the developers of Ciao for providing us with a copy of their framework. We thank Susan Sim for helpful feedback on earlier draft of this paper. We would also like to thank Bruno Lague and the Bell Datrix group for helping to motivate TAXForm by providing a reusable C++ extractor.

References

[1] Matt Armstrong and Chris Trudeau. Evaluating Architectural Extractors. In *Proc. of WCRE-98*. Honolulu, HI, October 12-14, 1998.

[2] Ivan Bowman and Richard C. Holt. Linux as a case study: its extracted software architecture. In *Proc. of ICSE-99*, Los Angeles, CA, May 1999. To appear.

[3] Peter Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9-36, March 1976.

[4] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, Ryan S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *Proc. of ICSM-95*, pages 66-75, Nice, France, October 1995.

[5] Datrix analysis tool. Bell Canada Quality Engineering Lab. http://www.iro.umontreal.ca/labs/gelo/datrix/

[6] P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proc. of ICSE-14*, pages 307-317, 1992.

[7] Ric Holt. An introduction to TA: The Tuple-Attribute language. Available at http://www-turing.cs.toronto.edu/pbs/papers/ta.html

[8] Ric Holt, editor. Conclusions from the Data Exchange Group Meeting. At CASCON'98 Nov. 30, 1998. http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/minutes98_11_30.html

[9] Ric Holt. Software Bookshelf: Overview and construction. Available at http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html

[10] Ric Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *Proc. of WCRE-98*. Honolulu, HI, October 12-14, 1998.

[11] Rick Kazman, S. Jeromy Carrière. Playing Detective: Reconstructing Software Architecture from the Available Evidence, *Journal of Automated Software Engineering*, 1998.

[12] Rick Kazman, S. Jeromy Carrière. View Extraction and View Fusion in Architectural Understanding. In *Proc. of ICSR-5*, Toronto, Canada, June 1998.

[13] Rick Kazman, Paul Clements, Len Bass, Gregory Abowd. Classifying Architectural Elements as a Foundation for Mechanism Matching. In *Proc. of COMPSAC-97*, Washington, DC, August 1997, pp. 14-17.

[14] Rick Kazman, Steven Woods, S. Jeromy Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proc. of WCRE-98*, pages 154-163, Honolulu, HI, October 1998.

[15] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse engineering of design components. In *Proc. Of ICSE-99*, Los Angeles, CA, May 1999. To appear.

[16] Timothy Lethbridge and Nicolas Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. University of Ottawa, Computer Science Technical report TR-97-07, 1997.

[17] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, E.R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proc. of IWPC'98*, Ischia, Italy, June, 1998.

[18] Hausi A. Müller. Criteria for Success of an Exchange Format. Workshop meeting, CASCON'98. 30 November, 1998. Available at: http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/criteria_muller.html

[19] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4), pages 181-204, December 1993

[20] Gail C. Murphy and David Notkin. Lightweight Lexical Source Model Extraction. *ACM Trans. on Software Engineering and Methodology* 5, 3 (July 1996): 262-292.

[21] Rational Rose 98i. Product Information. Rational Software Corporation. http://www.rational.com/products/rose/index.jtmpl

[22] David Rosenblum and Alexander Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119-134, April 1991.

[23] SNiFF+ v2.3. User's Guide and Reference, TakeFive Software. http://www.takefive.com, December, 1996.

[24] Margaret-Anne Storey and Hausi A. Müller. Manipulating and Documenting Software Structures Using SHriMP Views. In *Proc. of ICSM-95*, pp. 275-285.

[25] Margaret-Anne Storey and Kenny Wong. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proc. of WCRE-97*. Amsterdam, Holland, pages 12-21, October 6-8, 1997.

[26] Together/Enterprise. Product Overview. Object International Software. http://www.oi.com/

[27] Visio Enterprise 5.0. Product Overview. Visio Corporation. http://www.visio.com/products/enterprise/

[28] Kenny Wong. Rigi User's Manual. Version 5.4.1, July 10, 1996. Available at http://www.rigi.csc.uvic.ca/rigi/manual/user.html

[29] Steven Woods, Liam O'Brien, Tao Lin, Keith Gallagher, Alex Quilici. An Architecture for Interoperable Program Understanding Tools. *Proc. of IWPC'98*. Ischia, Italy, June 24-26, 1998.

[30] Alexander Yeh, David Harris, and M. Chase. Manipulating recovered software architecture views. In *Proc. of ICSE-19*, pages 184-194, May 1997.