Recovering UML Class Models from C++: A Detailed Explanation

Andrew Sutton, Jonathan I. Maletic[1]

Department of Computer Science, Kent State University, Kent Ohio 44242, U.S.A.

Abstract

An approach to recovering design-level UML class models from C++ source code to support program comprehension is presented. A set of mappings are given that focus on accurately identifying such elements as relationship types, multiplicities, and aggregation semantics. These mappings are based on domain knowledge of the C++ language and common programming conventions and idioms. Additionally, formal concept analysis is used to detect design-level attributes of UML classes. An application implementing these mappings is used to reverse engineer a moderately sized, open source application, and the resultant class model is compared against those produced by other UML reverse engineering tools. This comparison shows that the presented mapping rules effectively produce meaningful and semantically accurate UML models.

Keywords: Software engineering, reverse engineering, design recovery, program comprehension, UML class models

1. Introduction

The software industry has widely accepted and often uses UML (Unified Modeling Language) [Fowler'00] tools in forward engineering, but these tools are used less frequently during software maintenance and evolution. This is due to a number of reasons; foremost among these is that the manual recovery and maintenance of UML models is time consuming and costly. As such, UML models become stale while the source code continues to evolve. Although many UML modeling tools allow us to reverse engineer UML models from source code, they often perform poorly at this task. A case study of reverse engineering tools [Kollman'02] finds that many of these tools, despite advances in the research literature, continue to focus on producing the core elements of UML (i.e., simple class diagrams), but often fail to adequately represent design abstractions. This is problematic when recovered software models fail to accurately represent the abstract program semantics required for high-level program comprehension. This problem can be exacerbated by the fact that end users are typically unaware of the internal processes for producing the UML models.

[1] Corresponding author: Tel.: 330.672.9039
Email addresses: asutton@cs.kent.edu, jmaletic@cs.kent.edu

This is to say that these tools do not disclose their mechanisms for reverse engineering, which can lead to results that do not meet the end-user's expectations.

Although this study [Kollman'02] concludes that these tools provide reliable functionality, the resulting models are anything but consistent. For example, Microsoft Visio is incapable of reverse engineering associations, Visual Paradigm creates dependencies when associations are appropriate, and Rational Rose C++ Modeler creates only aggregate associations (open diamonds in UML). The primary reason for these inconsistencies is the sizeable semantic gap between UML and C++. Although this gap is quite wide, it is by no means unbridgeable. Unfortunately, commonly used reverse engineering tools are closed source systems and provide little information about how UML models are created from C++. This leaves developers to speculate about rationale for the application's logic. As such, there is no standard "bridge" between C++ and UML, and all reverse engineering tools tend to build their own.

We address this problem by defining a set of mappings for the reverse engineering of UML class models from C++ source code [Sutton'05, Sutton'05]. These mappings employ a combination of C++ syntactic and semantic information along with domain knowledge of programming conventions, idioms, and reuse libraries to produce semantically accurate UML class models. Many of these mappings extend and integrate techniques presented in the literature on this topic. As part of these mappings, a sophisticated information analysis technique (formal concept analysis) is applied to the UML model to recover design-level attributes of classes rather than re-document member variables.

These mappings are implemented in a reverse engineering tool, *pilfer*, which is used to evaluate the relevance of the defined mappings by reverse engineering a moderately-sized C++ application. The model produced by *pilfer* is compared against those produced by other tools in order to validate the accuracy and completeness of the defined mappings. Because performance is an important aspect of reverse engineering tools, we also compare *pilfer*'s run time performance against these tools.

This paper is organized as follows. Section 2 provides a more detailed context of the problem being addressed. Section 3 describes rationale and tradeoffs for each mapping. Section 4 describes the implementation of *pilfer*. In section 5, we present a comparison of models generated by *pilfer* and other reverse engineering tools. Section 6 describes work related to this topic and section 7 presents our conclusions and future work.

2. Reverse Engineering Analysis

Design recovery is the process of recovering design decisions, abstractions, and rationale from a program's source code [Chikofsky'90]. Design recovery directly supports program comprehension through reverse engineering. Figure 1 depicts the architecture of a technology stack used in reverse engineering to recover program designs. This technology stack is motivated in part by the *Rigi* reverse engineering environment [Storey'97, Wong'95] and the DMS program

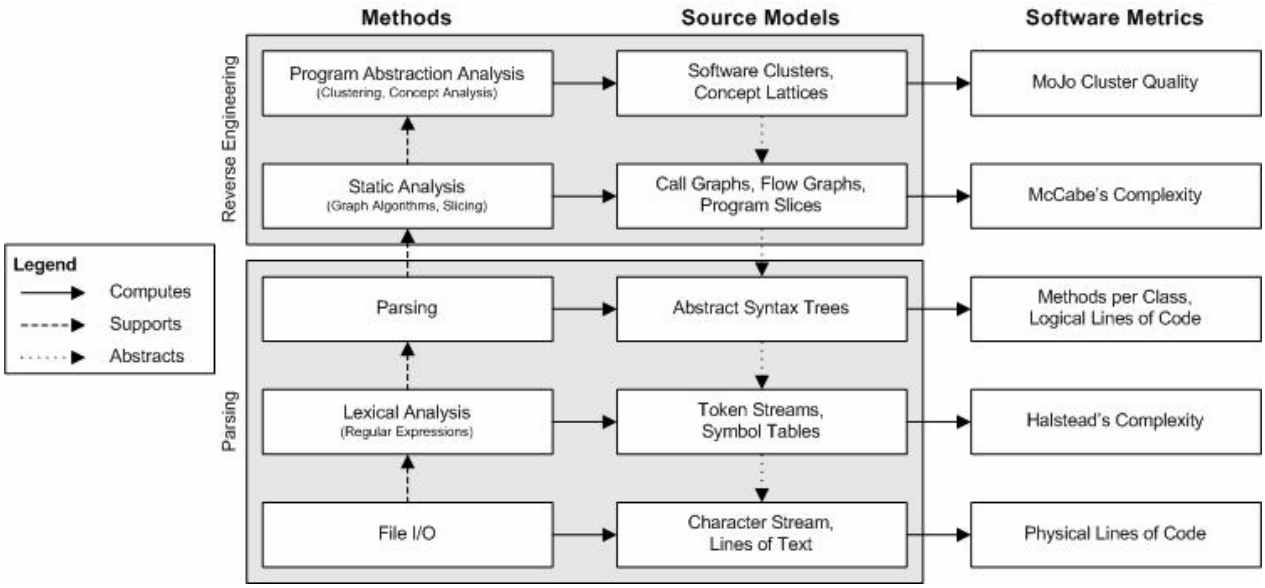

Figure 1. A reverse engineering stack produces increasingly abstract source models based on layered processes and analyses. Software metrics are often computed for source models at different levels of

Table 1. A list of concepts being reverse engineered, the required level of analysis to perform that task including domain knowledge and the degree of ambiguity associated with each task.

UML Concept to be Reverse Engineered	Required Analysis			Domain Knowledge	Degree of Ambiguity	Description of Reverse Engineering Task
	Parsing	Semantic Analysis	Static Analysis			
Entities						
Classes	X				None	
Interfaces		X		X	Med	Distinguish interfaces from classes
Data types		X		X	Med	Distinguish data types from classes
Attributes	X				None	
Design-level attributes			X	X	High	Determine design-level attributes
Read-only attributes		X		X	Low	Find attributes with only accessors
Attribute type		X		X	Med	Correctly resolve type references
Multiplicity			X	X	Med	Identify multiplicity of attributes
Ordering			X	X	Med	Identify ordered and unordered containers
Parameters	X				None	
Parameter direction			X	X	Med	Identify in, out or inout parameters
Relationships						
Association		X			High	Find attributes constituting associations
Aggregation semantics			X	X	High	Determine object lifetime
Realization		X			Med	Distinguish inheritance from implementation

analysis system [Baxter'04]. It integrates the technologies used in reverse engineering such as static analysis, concept analysis and software clustering. More precisely, Figure 1 illustrates an architecture (or reference model) for a reverse engineering environment. This architecture is composed of layered analysis methods and models. Each method computes a model of the system that is consumed or used by higher level analyses (e.g., parsing yields an AST that can be used to compute control flow graphs). Moreover, there are a number of potential metrics (those shown are only a small subset of software metrics) that can be computed from the models at all layers of this architecture.

Much like the computation of metrics, *mapping rules* (or simply mappings) can be defined at any level of abstraction. Although these mappings can be used to produce any number of artifacts, here we specifically envision them being used to produce UML models. For example, simple mappings from C++ to UML produce UML classes from the C++ classes found in source code. However, these simple mappings – those most often implemented in common UML reverse engineering tools – are fairly naïve. They rarely embed more than a rudimentary knowledge of programming language semantics or libraries, nor do they allow users to embed their own specific domain knowledge into the reverse engineering processes. Moreover, with a limited set of analysis methods, these tools are often incapable of producing anything more than visually redocumented source code. Reverse engineered artifacts of this nature are often too detailed to be of any great use to the casual reader. The information they relate is easily attainable from the source code, or through a source code redocumentation tools such as Doxygen or JavaDoc.

The use of domain knowledge within these mappings provides a significantly broader range of functionality. Domain knowledge can includes programming style conventions (e.g., idioms, patterns, and even method naming), knowledge of reuse libraries, and even information from the problem domain. Additionally, these mappings can leverage other abstract source models (e.g., control flow, data flow, and call graphs) to assist in the mapping process.

Table 1 describes the UML concepts [OMG'03] to be reverse engineered from source code. These concepts are representative of current deficiencies in reverse engineering tools as noted in [Kollman'02] and through our practical experience with a number of reverse engineering tools. Although the number of tasks required to fully reverse engineer C++ programs is significantly longer, we are interested in defining mappings for a small subset of those.

The columns in Table 1 are described as follows: the *required analysis* column describes the analysis method required to accomplish each task. This classification is based primarily on experience with C++ parsing technology and the implementation of different reverse engineering methods. Parsing implies that the results could be obtained through the abstract syntax tree (AST) of the program, semantic analysis means that the application must rely upon specific C++

semantics in order to perform the task, and static program analysis (or simply static analysis) indicates the need for more sophisticated analysis methods (e.g., data flow analysis). The *domain knowledge* column indicates whether or not domain knowledge will contribute to the completion of the task. Here, domain knowledge is taken to include knowledge of the problem domain and (primarily) the solution domain of an application and its implementation. For example, knowledge of programming idioms, design patterns, and code concepts (e.g., STL template concepts) are potentially useful when recovering the semantics source code. Tasks in which the application of domain knowledge plays a role are marked. The *degree of ambiguity* describes how consistently different tools perform the task. Low ambiguity implies that tools produce mostly consistent models, whereas high ambiguity tasks result in widely varying results. The degree of ambiguity was determined by considering the number alternative design-level semantics for a given C++ declaration. For example, identifying UML associations is highly ambiguous; this becomes evident when comparing the results of different reverse engineering application. The *description* column provides additional details for the reverse engineering task.

Note that we have not included methods (member functions) in this table. Most reverse engineering tools typically have little difficulty identifying the methods of a class. Here we are interested in which methods are actually produced in the UML model. Most tools include constructors, destructors, and overloaded operations in the reverse engineered classes. However, we see these as implementation details (i.e., language integration features) that only serve to pollute the resultant UML model. We recommended allowing the user to decide the relevance of those methods to their comprehension requirements.

3. Mappings for Reverse Engineering

In this section, we define mappings for UML reverse engineering tasks with a degree of ambiguity of medium or higher in Table 1, or those that are often seen as difficult or having potential ambiguities in their mappings. The mappings defined herein are heuristics based on syntactic and semantic features rather than deterministic analyses.

3.1. Identifying Types of Classes

The distinction between *classes*, *interfaces*, and *data types* is semantically important in UML. Taken as a whole, these elements are described as *classifiers*, or named modeling elements that a) have properties and behaviors and b) can participate in generalization relationships. However, the treatment of these elements in models can vary greatly. For example, UML does not allow associations between classes and data types, and only classes can realize interfaces. Unfortunately, C++ does not provide a rich enough vocabulary to easily distinguish these classifiers through simple parsing. Moreover, it is not always obvious whether a C++ class is a UML data type or interface. Let us now address each of these separately.

3.1.1. Interfaces. Our method for identifying interfaces is fairly straightforward. Our definition of *interface* is borrowed from other OO languages, namely Java and C#. We define a C++ interface as a class that defines only public, pure virtual methods, declares no member variables, defines no constructors or destructors, and, if derived, the base classes must also be interfaces.

```
struct IList
{
  virtual int add(Object *) = 0;
  virtual void remove(Object *) = 0;
  virtual bool contains(Object *) = 0;
  virtual void clear() = 0;
};
```

Figure 2. The *IList* class is an interface, providing only public abstract methods, and defining no state or behavior.

A number of these restrictions derive from the notion that interfaces specify a *contract* rather than program behavior. For example, a class implementing a method associates behavior with that class. Moreover, a class declaring member variables associates state information with the class. Obviously, if a class declares no member variables then it has no need of specialized constructors or destructors. Finally we restrict interfaces to being derived only from other interfaces in order to align our definition with common OO models.

The code in Figure 2 shows a class, *IList*, that meets our criteria for interface declarations. It defines only public pure virtual methods, and defines member variables, constructors, or destructors. Although we can use this knowledge to effectively distinguish a “true” C++ interface from other abstract classes, there are some tradeoffs in its usage. First, our definition of a C++ interface is fairly restrictive and may not align well with the common convention of treating abstract classes as interfaces. As such, this mapping may produce some unexpected results for some developers. Second, earlier

versions of UML greatly restrict the usage of interface elements in associations (i.e., they do not participate in associations). These restrictions have been relaxed in newer versions.

3.1.2. Data Types. In UML, a data type is one that is identified only by its value such that two instances having the same value are said to be the same instance. Instances of classes however, have a distinct identity and two objects with the same state, are not necessarily considered to be the same objects. Typical examples of data types include programming language primitives such as integers, Boolean values, and enumeration values. A string class is also an example of a data type. In order to identify data types in C++, we have to rely heavily on how classes are used in a program. Specifically, we use a class's construction, copy, and assignment semantics in order to identify it as a data type.

```
struct Complex
{
  double mReal;
  double mImaginary;
};

struct String
{
  String();
  String(const String &);
  String &operator =(const String &);
private:
  char *mText;
};
```

Figure 3. The *Complex* class relies on the compiler to supply its copy, construction and assignment semantics. The *String* class overloads these methods to provide specialized copy and assignment semantics.

Our definition of a C++ data type encompasses two distinct variations. A class that implements a public default constructor, a copy constructor, and assignment operator is a data type. In this case, the *String* class in Figure 3 explicitly implements default construction, copy construction, and assignment semantics that will allow the class to behave like a POD (Plain Old Data) type. Also, a class that implements no constructors or assignment operators is also a data type (such as the *Complex* class). In this case, the developer is relying on the compiler to supply defaults for these methods. Note that we do not consider destructors in the classification. Destructors add little to the design-level semantics of the class because *all* classes have destructors – either implicit or explicit. As such, using the destructor in classification may lead to ambiguity when classifying C++ classes.

Automating this mapping might lead to cases where classes are misidentified as data types, especially in cases where the author intentionally implements the constructor, copy constructor and assignment operator. However, we feel that these cases are rare, and might even represent extraneous functionality for those classes (e.g., dead code or poor class design).

3.2. Identifying Attributes

Typical reverse engineering tools correlate UML attributes with a class's member variables. However, in UML, attributes are used somewhat differently. Typically, an attribute reflects a facet of a class's interface that can be read or written rather than representing the implementation details of a member variable. This is to say that UML attributes more likely correspond to instances of the property idiom rather than the member variables of the class. This is more appropriate for reverse engineering tools because it represents a more abstract view of a class rather than its implementation details. Programming languages such as C# provide syntactic features that allow the explicit declaration of such properties, but C++ does not.

Our method for identifying attributes of classes is based on the collection of accessors and mutators of a C++ member variable. For the purpose of this discussion, we define an accessor as a constant method that returns a member variable of a class. A mutator is a method that writes the value of a member variable. Accessors and mutators are grouped by the member variables on which

```
class ModelElement
{
public:
  const string &name() const;
  void name(const string &);

  const guid &id() const;

private:
  int mRefCount;
  guid mId;
  string mName;
};
```

Figure 4. The *ModelElement* class has three member variables but defines only two modeled properties: its name and unique id. The reference count is a detail of the implementation.

Table 2. Example mappings from groups of C++ member-function declarations to UML attributes.

C++ Member Function Declaration	UML Type
`const Foo &foo() const;` `void setFoo(const Foo &);`	Read-write property, "foo"
`Foo *foo() const;` `void setFoo(Foo *);`	Read-write property, "foo"
`const Foo &foo() const;`	Read-only property, "foo"
`Foo *foo() const;`	Read-only property, "foo"

they operate. A read-only property can be identified by an accessor returning a member variable. A read-write property can be identified by the presence of an accessor and a set of mutators. We define writable properties as having a *set* of mutators because the property's interface could support collection semantics (e.g., add, remove, and clear). Examples of mapping rules are show in Table 2.

Figure 4 shows a class with two model-able properties: *id* (read only) and *name* (read-write). These properties are derived by examining accessor and mutator methods and their relationship to member variables of the class.

However, this method of detecting attributes of C++ classes is not without fault. Attempts to automate this detection without more sophisticated analysis techniques will almost certainly lead to the detection of false positives. This can have wide ranging effects if behaviors (i.e., methods) of a class are mis-modeled as UML attributes. However, a developer performing this task manually should have some intuition about what features of the class are properties and which are operations, allowing them to disambiguate cases where the function definitions are unclear.

Table 3. Mappings from C++ type specifications to UML types discard qualifiers and pointer, reference, and array tokens.

C++ Type Declaration	UML Type
`Foo foo;`	Foo
`Foo *foo;`	Foo
`Foo **foo;`	Foo
`const *Foo;`	Foo
`Foo &foo;`	Foo
`const &Foo;`	Foo

Table 4. Transitive type mappings from C++ template typed declarations use the inner-most template arguments to construct UML type information.

C++ Declaration	UML Type
`list<Foo>`	Foo
`set<Foo *>`	Foo
`const stack<Foo> &`	Foo
`queue< auto_ptr<Foo *> >`	Foo

3.2.1. Attribute Type. Although it is relatively easy to determine the type of a member variable in C++, that type does not always map directly to UML. For example, UML provides no syntax for modeling pointer or reference types, and many common C++ type qualifiers (*const*, *mutable*, and *volatile*) can have little or no meaning because typed elements in UML are simply references to classifiers. No additional information is modeled in the specification of type.

To this end, we define a simple mapping for type resolution. We define the type of a modeled attribute to be the type reference in a C++ type expression. The type reference can be obtained by stripping out all pointer, reference, array symbols, and any qualifiers in the expression. Examples are shown in Table 3.

In addition to the simple C++ type expression mapping in Table 3, we also need to deal with more complex template type expressions. Templates such as containers and smart pointers in the STL (Standard Template Library) are used frequently in C++ programs, but do not necessarily contribute to type information in a UML model. Instead, they embed semantics about the association between classes, but not about the types of the classes participating in that association. Such classes exhibit a transitive property for the collaborating classes. We consider these classes to be transitive in nature if they satisfy a condition of transitive containment. That is to say, "class *A* contains class *B* which contains class *C*, and therefore class *A* contains class *C*." In this case, class *B* is either a container or smart pointer. It is the summary statement that "class A contains class C" that is more appropriate in UML type specifications. The relevant type information can be extracted from template declaration by extracting its innermost arguments. The mapping for extracted type information from simple C++ type specification also applies. Examples mappings are shown in Table 4. As such, automating this mapping requires a great deal of embedded domain knowledge. An application performing this task needs to know in advance which classes admit this transitive containment property and which template parameters correspond to the appropriate type information.

Table 5. Mappings from C++ declarations to UML multiplicity ranges depend on pointer, reference, and array symbols associated with the type reference. Template classes can also contribute multiplicity information.

C++ Declaration	UML Multiplicity Range
`Foo`	1..1
`Foo *`	0..*
`Foo []`[1]	0..*
`Foo *[]`[2]	0..*
`Foo [n]`[3]	*n*..*n*
`Foo *[n]`	0..*n*
`Foo **`	0..*
`Foo &`	1..1
`list<Foo>`	0..*
`auto_ptr<Foo *>`	0..1

[1] The expression *Foo []* is only usable in formal parameter lists and is semantically equivalent to *Foo **.
[2] The expression *Foo *[]* is only usable in formal parameter lists and is semantically equivalent to *Foo* **.
[3] Where *n* is a constant, integral value.

3.2.2. Multiplicity. Multiplicity defines the allowable number of instances of an attribute and is expressed as a

range between an lower and upper bound (e.g., 0..*). The multiplicity of attributes can be difficult to detect. There is no set of simple rules that readily describe a mapping of declarations to multiplicities. Fortunately, there are several indicators in C++ that can help us approximate the multiplicity of an attribute – especially pointers, array brackets, and transitive classes. Table 5 lists the mapping rules between C++ type declarations and UML multiplicity ranges.

Note that the only multiplicity ranges that can be unambiguously identified are those where a) only a single instance is declared, b) a reference to a single instance is declared, or c) a fixed-size array is declared. In all other cases, we cannot accurately identify either the lower or upper bounds of the range. This is due to the ambiguity of C++ declarations. For example, we might typically expect a pointer declaration (i.e., *Foo* *) to represent a single object, but C++ defines no difference between this and a C-array of *Foo* objects. Additionally, we can use knowledge of containers and smart pointers to extract multiplicity semantics.

3.2.3. Ordering. Because many attributes represent the containment of multiple instances, the UML metamodel provides the ability to describe ordering semantics for containers. UML defines two types of ordering for attributes: *ordered* and *unordered*. These simply specify whether the containing attribute stores instances sequentially (e.g., a list or vector) or otherwise (e.g., a set). To date, there is no good static analysis method that can accurately recover the ordering semantics of containers. This is due to the fact that the storage mechanisms are woven throughout the implementation of various data structures. Fortunately, we can use the declarative semantics of arrays and the use of container classes to aid in this mapping. Mappings for container ordering semantics are shown in Table 6.

Table 6. Mappings from C++ declarations to UML orderings depend primarily on the semantics of abstract data types (e.g., set or vector).

C++ Declaration	UML Multiplicity Ordering
`Foo, Foo *`[1]	ordered
`Foo []`	ordered
`Foo *[]`	ordered
`vector<Foo *>`	ordered
`list<Foo>`	ordered
`deque<Foo *>`	ordered
`set<Foo>`	unordered

[1] Member variables with single or optional (1 or 0..1) multiplicity are typically described as *ordered*, which is the default ordering given by the UML specification.

The mappings for ordering semantics are easily derived from information about member variable type declarations. C-array and C-vector declarations are always allocated with sequential memory storage. The ordering semantics of containers are intrinsic to their data type.

We might note that the *ordered*/*unordered* attribute semantics of UML do not map precisely onto the concepts defined by the STL. An *ordered* attribute corresponds to the STL *Sequence* concept in that contained elements are arranged in a strict linear order. Examples include array, vectors, deques, and lists. On the other hand, the *Sorted Associative Container* concept has no equivalent semantics in the UML specification although the possibility of a vendor-specific extension for sorted attribute types is mentioned.

3.3. Identifying Parameter Direction

UML Parameters share some features with UML attributes (e.g., type resolution) and as such can be reverse engineered using similar mappings. However, UML parameters also encode information about how their values are transmitted to (and from) an operation. This information is called the parameter's direction kind. UML defines four directions for parameter passing: *in*, *out*, *inout* and *return*. The direction determines whether or not the parameter will be used as an input to an operation, an output of the operation, both (i.e., in the case that an operation that changes the state or value of an input), or as a pure return value. Because C++ does not provide us with enough declarative granularity for this mapping, we define a list of mappings based on the parameter's type and its qualifiers. These mappings are shown in Table 7.

Table 7. Mappings from C++ parameter declarations to UML parameter direction rely upon pointer, reference, and const qualifiers in the type specification.

C++ Declaration	UML Parameter Direction
`Foo`	in
`Foo &`	inout
`const Foo &`	in
`Foo *`	inout
`Foo **`[1]	out
`const Foo *`	in

[1] While it is possible to use parameters of this type as inputs, API designers often use pointers to pointers to store the outputs of pointer manipulation (e.g., in-place memory allocation).

In C++, arguments are passed to methods either by reference or by value. Because pass-by-value parameters result in local copies of the supplied arguments, they can easily be identified as *in* parameters. However, if the parameter is

passed by reference, we need to examine the const-ness of the declared parameter. If a parameter declaration includes the *const* keyword, then it can be modeled as an *in* parameter. Otherwise, it can be modeled as an *out* parameter.

3.4. Identifying Associations

Although UML associations are most often used to represent "has-a" relationships, they are sometimes employed to model semantic relationship between two different classes. C++ does not provide a syntactic concept for modeling these semantic links. However, it is fairly easy to discern the "has-a" relationships from the member variable declarations of a class. Our method for identifying associations in C++ relies heavily on the correct identification of C++ classifier types and declarative type information. We define a C++ association as a member variable with a type reference to a modeled UML class, but not a data type. We restrict classes and interfaces from being associated with data types because the semantics of that particular relationship are wholly encapsulated in the fact that the member variable is modeled as an attribute. Note that the derivation of the type reference used in this mapping must follow the mapping rules for type resolution.

```
class ModelElement
{
  Namespace *mNamespace;
};

class Namespace
{
  set<ModelElement *> mOwnedElement;
};
```

Figure 5. The *ModelElement* and *Namespace* classes participate in associations according to our heuristic. They both define member variables referencing another class.

The code in Figure 5 shows two classes participating in associations. The *mNamespace* member is an obvious candidate. In the case of *mOwnedElement* we extract the inner type of the *set* member to define an association between *Namespace* and *ModelElement*.

3.4.1. Aggregation. An association's aggregation kind defines lifetime semantics for instances contained through the relationship. UML defines three types of containment semantics for associations: *none*, *aggregate*, and *composite*. The only types of aggregation kind that can be derived from the C++ grammar are *aggregate* and *composite*. The *none* variety of aggregation kind represents purely semantic links between classes and is of little interest in this context.

Determining the aggregation kind of a property is difficult because C++ provides very little language support for declaring or embedding shared and composite association semantics. For example, a member variable pointing to another object could be a composed member of its enclosing class (as in the private implementation idiom), or it could simply be stored for convenience and shared between a number of other objects. The use of smart pointers allows the developer to embed specific lifetime semantics into a program. Recognizing these beacons can drastically reduce the amount of effort spent deducing the proper aggregation kind of these elements.

Much like the determination of multiplicity, the determination of aggregation kind has no simple set of rules. C++ provides some declarative information that can be used in the mapping such as pointer, reference, and array symbols. Also, transitive classes such as containers and smart pointers can be used in this mapping. Table 8 shows mappings of declared member variables types into UML aggregation semantics. Additionally, we require the attribute being analyzed to participate in an association before aggregation semantics can be defined.

Unlike smart pointers, container classes can have varied semantics. Most STL container classes define transitive aggregation semantics. This is to say that aggregation kind can be deduced from the inner type reference in the template expression according to the simple type expression mappings in Table 8. Specific domain knowledge of these classes must be used to deduce aggregation semantics.

Table 8. Mappings from C++ declarations to UML aggregation semantics rely upon pointer, reference, and array symbols within the type declaration. Additional information can be acquired from transitive classes.

C++ Member Variable Declaration	UML Aggregation Kind
`Foo`	composite
`Foo *`[1]	aggregate
`Foo &`[2]	aggregate
`Foo []`	composite
`Foo *[]`	aggregate
`auto_ptr<>`[3]	composite
`boost::scoped_ptr<>`	composite
`boost::shared_ptr<>`	aggregate

[1] We consider any level of indirection to be aggregation. Multiple levels of indirection generally imply the use of arrays, C-vectors or C-matrices.

[2] Although it is typically rare to define classes with reference member variables, it is possible. This requires an instance of the referenced type be passed as a parameter to a suitable constructor, and that the lifetime of the referenced object must exceed that of the container.

[3] The auto_ptr<> class allows ownership to be transferred so we cannot determine composition without additional analysis.

3.5. Identifying Realization

In UML, inheritance is represented by generalization relationships, which can be easily determined through C++ inheritance specifications. However, the implementation (realization) of interfaces poses a slightly different problem. Although the syntactic mechanisms for realization inheritance are the same in C++, UML provides additional syntax that can be used to express interface semantics. In UML realization is a dependency between a class and an interface, expressing the fact that the class implements the contract specified by the target interface.

Our method for recognizing realization relationships relies on the ability to correctly determine classifier types. If a class implements *all* the abstract methods of an interface, then a realization relationship can be created between the class and the interface. Partial realization only results in an abstract base class derived from the interface. We might note that realization is not a replacement for generalization. The class is still derived from an interface, so the generalization relationship could also be modeled.

4. Implementation

The *pilfer* reverse engineering tool is currently implemented in the Python programming language. This was chosen for a number of reasons. First, it allows the application to be built and modified quickly, allowing developers to modify or experiment with the given mappings. *pilfer* leverages two key technologies to implement its reverse engineering capabilities: srcML[2] and the Open Modeling Framework (OMF) [3]. These technologies are explained in sections 4.1 and 4.2 respectively. In addition to the software used to implement the source code parsing and UML modeling, *pilfer* implements the mappings described above. Additional analysis components are also present in the implementation – specifically for the detection of attributes from C++ member variables.

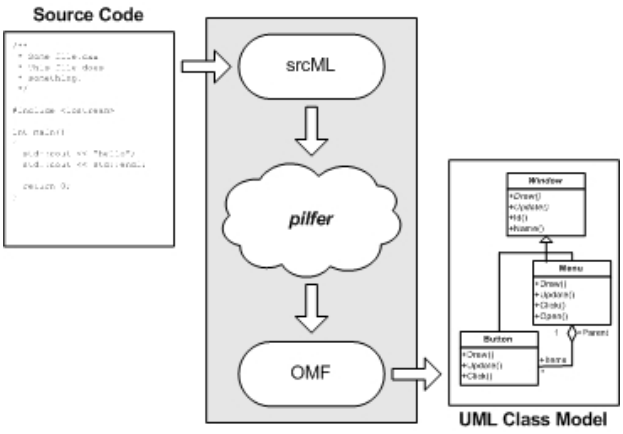

Figure 6. The *pilfer* tool architecture. Source code is transformed into srcML, mapped into UML, and output as XMI.

The architecture and workflow for *pilfer* is shown in Figure 6, which describes the interaction between the different components of the tool chain. Specifically, srcML is used to produce an XML representation of the source code. *pilfer* uses the srcML output to construct an abstract semantics graph (ASG), perform semantic analysis and execute the mappings over the resulting graph. Finally, UML model elements are constructed and serialized through the OMF. This section describes these components in detail.

4.1. srcML

Rather than implement another C++ parser specifically for the purpose of implementing *pilfer*, we have instead turned to an existing, successful application for fact extraction. srcML (SouRce Code Markup Language) [Collard'03, Collard'02, Maletic'02] is an XML representation that supports document and data views of source code. The source code document is preserved within the XML format by retaining all of the lexical information (e.g., white space, preprocessor directives and comments) within the original file. A data view is provided by the srcML format by the addition of XML elements representing the syntactical structures of the C++ programming language (e.g., functions, classes, statements, etc.). A srcML translator, *src2srcml*, has been developed to transform C++ into its srcML representation.

We use srcML as a parsing platform for a number of reasons. Because srcML is an XML format, we can leverage any number of XML APIs to query or explore the program during reverse engineering. For example, we use a DOM tree (actually *libxml2*) to walk the AST of source files and produce a C++-specific model. Moreover, srcML is a lightweight (i.e., coarse-grained) markup language as opposed to, say, that of GCCXML[4]. This means the size of intermediate files is only somewhat larger than the original source files (usually by a factor of less than 5) allowing pilfer to quickly process the marked up source code. Intermediate formats like GCCXML encode the entire AST in XML and

[2] See www.sdml.info/projects/srcml/ for details on srcML.
[3] See www.sdml.info/projects/omf/ for details on the OMF.
[4] See www.gccxml.org/ for details on GCCXML.

produce files that can be several orders of magnitude greater than the original source code, and analyses based on these products will run much slower.

4.2. The Open Modeling Framework

There are few viable solutions to producing UML models without tying into an existing, monolithic modeling application. The Open Modeling Framework is a C++ implementation of both the Meta-Object Facility (MOF) and UML metamodels. In this regard, it is similar to both the Java Metadata Interface (JMI)[5] and the Eclipse Modeling Framework (EMF)[6], but adds comprehensive support for UML modeling. Moreover, the OMF provides a model-centric view of UML that focuses on models and their components rather than the content or management of diagrams. This allows developers to focus on the inspection, validation, or manipulation of models rather than trying to dovetail analysis-intensive tools to application framework interfaces provided by the existing graphical applications such as Rational Rose, Microsoft Visio and Visual paradigm. This feature has significant impact in the domain of reverse engineering tools where the focus is on source code analysis and model construction rather than the rendering of user interfaces and layout UML diagrams. Additionally, scriptable Python interfaces provided by the OMF enable integration with other (non-UML) technologies such as grammar parsers, databases, and XML.

We use the OMF as *pilfer*'s internal representation of UML. Because the OMF supports reading and writing of multiple versions of XMI, the resulting models can (theoretically) be exchanged between modeling applications for visualization – although the interoperability of XMI dialects is a well-documented problem [Jiang'03].

4.3. Implementing *pilfer*

The primary purpose of the *pilfer* software is to provide a framework that allows the development of flexible and replaceable mappings between C++ and UML. *pilfer* does this by iterating over the AST provided by srcML inputs and constructing an internal, C++-specific, abstract syntax graph (ASG). Additionally, *pilfer* implements lightweight semantic analysis of this graph. Specifically, *pilfer* augments the ASG with C++ language concepts such as the instantiability of the class, whether or not is default or copy-constructible, assignable, etc. Many of these concepts are critical to the implementation of UML mappings.

Once this model is obtained, the process of mapping it to UML is fairly straightforward. It begins by iterating over the elements of the model and executing specific mappings based on the type of element. Each mapping is implemented as a single function that performs one or more responsibilities. For example, the UML class mapping examines the list of constructors, the destructor, and operators to determine if the class is a data type. If not, it also examines the list of methods to determine whether or not the class is a UML interface. Finally, the mapping will create a UML element corresponding to the deduced classifier

Many of the other mappings provided with the *pilfer* software are required to analyze type information to deduce UML semantics. These are implemented using simple string matching techniques. For example, the multiplicity of an attribute can be easily determined by searching for '*', '[', or ']' characters within the type specifications. In fact, this technique is applied to many mappings implemented by pilfer.

Deducing information from template classes is somewhat more complicated. In order to determine the semantics of a template instance, *pilfer* creates a "template instantiation tree". This is built such the leaf nodes are represented by non-template types. A traversal of the tree yields the names of templates that contribute to the semantics of the declaration and the underlying types of the declaration as well. The template classes found within the tree are used to "guide" the traversal and how the semantics are extracted from the tree. For example, the *std::list* class uses its first template parameter as its value type (the type being stored), whereas the *std::map* class uses the second parameter as its value type. We might also note that template classes may provide somewhat ambiguous type and semantics information. The *std::pair* class is one such template that resists the application of type and semantics as described above.

Additionally, *pilfer* allows the use of configuration options to control several of its implemented mappings. *pilfer* supports configuration options for the detection of design-level attributes (discussed in section 4.4), the interpretation of pointer multiplicities, and the recovery of composite associations. The options for controlling attribute recovery allow the user to activate (or deactivate) the activity. When not activated, *pilfer* will simply modify member variables as UML attributes. The option for interpreting the multiplicity of pointers, will create 0..1 multiplicities for pointers. It is common practice to use pointers to reference single objects rather than allocate C-arrays so 0..* multiplicities may be

[5] See java.sun.com/products/jmi/ for details on the JMI.
[6] See www.eclipse.org/emf/ for details on EMF

inappropriate. The use of this option depends on knowledge of the programming conventions used in the target system. Finally, the option for controlling association generation allows the *pilfer* to restrict the interpretation of associations. In some cases, it may be useful to limit the recovery of associations to only member variables that are references or pointers. If this option is used, non-referential member variables are not considered when attempting to determine if an attribute should be modeled as an association. However, this is disabled by default.

4.4. Detecting Design-Level Attributes

In many cases, reverse engineering tools create too much information. This is to say that the mappings implemented by such tools generate detailed listings of a class's implementation rather than attempting to present more abstract, design level views of the system. To address this issue, *pilfer* can automatically detect design-level attributes of classes. To do this, *pilfer* will only model member variables that are associated with a set of accessor and mutator methods. Moreover, the methods associated with that attribute will be excluded from the list of methods being modeled.

The implementation of this method employs a sophisticated information analysis technique – formal concept analysis [Eisenbarth'01, '03, Lindig'97]. Here, concept analysis is used to find groupings of trivial accessors and mutators in C++ methods, allowing us to deduce the design-level attributes from the resulting concept lattice. The use of concept analysis allows us to reduce the information being reverse engineered into UML, and significantly improves the quality and readability of resulting class diagrams. Moreover, there is little to no loss of design-level semantics in the resulting models. This is to say that performing this reduction on a C++ implementation results in semantically appropriate UML model.

Concept analysis operates on a relationship between a set of objects and the set of attributes that each object is described by. The algorithm computes the maximal set of attributes associated with a set of objects. Each maximal subset of objects and attributes is called a *concept*. Concepts are related by the attributes that each object shares, creating a lattice of related concepts.

In order to identify abstract attributes of C++ classes, we use concept analysis to identify cohesive units of functionality clustered around attributes. For our purposes, we define the attribute set as the set of member variables declared within a class, and the object set as the member functions defined within the class. The only relationship used is that in which a method in the object set reads or writes an attribute value in the attribute set. In other words, we are interested in which member functions use which member variables.

The examples in this section are based on the HippoDraw application (discussed in section 5). Table 9 shows a list of member variables and functions from HippoDraw's *Range* class and their usage table. This table was constructed for this example by examining which member functions used which member variables and removing the "m_" from each member variable. No distinction is made as to the actual type of usage (read, written, returned, or passed as an argument). This attribute usage is then used to produce a concept lattice according to the concept analysis algorithm. Each concept consists of an extent (the objects in each concept) and its intent (the attributes in each concept) that each function uses. The extent and intent of each

Table 9. The usage of member variables (attributes) by member functions (objects) in HippoDraw's *Range* class.

		Member Variables (Attributes)			
		min	max	pos	empty
Member Functions (Objects)	low	X			
	setLow	X			
	high		X		
	setHigh		X		
	pos			X	
	setPos			X	
	setRange	X	X	X	
	setLength	X	X		
	includes	X	X		
	excludes	X	X		
	fraction	X	X		
	setIntersect	X	X	X	
	setUnion	X	X	X	
	setEmpty				X
	numberOfBins	X	X		

Table 10. Each concept in the resulting lattice consists of a maximal set of objects and attributes.

Concepts	Object and Attribute Sets
top	{{low, setLow, high, setHigh, pos, setPos, range, setRange, setLength, includes, excludes, fraction, setIntersect, setUnion, setEmpty, numberOfBins}, ∅}
c0	{{low, setLow}, {min}}
c1	{{high, setHigh}, {max}}
c2	{{pos, setPos}, {pos}}
c3	{{setEmpty, empty}}
c4	{{setLength, includes, excludes, fraction, numberOfBins}, {min, max}}
c5	{{setRange, setUnion, setIntersect}, {min, max, pos}}
bottom	{∅, {min, max, pos, empty}}

concept in the *Range* class are shown in Table 10. Information in this table is used to generate the lattice shown in Figure 7. Because the extents are required to be maximal collections of objects sharing common attributes, the resulting concepts represent potential cohesion between member functions around a set of member variables. Figure 7 illustrates the relationship between member functions using sets of common attributes. It can be seen that concepts in the first tier of the lattice (i.e., those labeled *max*, *min*, *pos*, and *empty*) actually represent accessors and mutators for the given member variables. Using this lattice, we can easily define a heuristic for automatically identifying the clusters of mutators and accessors of a member variable. Essentially, we treat any concept in the first tier with exactly one member variable and one or more functions as an abstract class property. When such an attribute is found, it is modeled as a UML attribute. The member functions representing the set of accessors and mutators are removed from the list of UML operations to be modeled. Note that member variables that do not meet these criteria are not modeled in UML because they represent implementation level attributes.

The concept analysis capability is implemented as an optional analysis component of *pilfer*. This is to say that *pilfer* can be run with or without using this technique, the latter producing implementation views of the source code. Specifically, *pilfer* traverses the UML model and runs the concept analysis algorithm on every UML class, data type, and interface. The implementation is built around Lindig's implementation [Lindig'97] via its python bindings. When used, *pilfer* can generate *dot* (GraphViz[7]) files describing the concept lattices such as the one pictured in Figure 7.

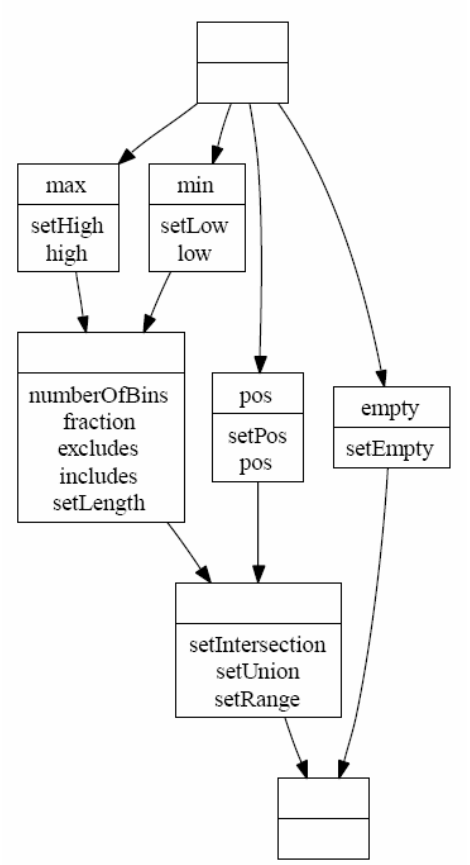

Figure 7. This concept lattice represents the usage of member variables by functions. Only the sub-concepts of the top concept define abstract properties.

Experiments with this technique have shown that this approach is very successful at finding design-level attributes. In fact, we found that this approach also successfully reduces delegate interfaces of composed member variables. This is to say that it can eliminate portions of a class interface that simply delegate to a contained member variable. We conducted two experiments to validate the applicability of this technique. These experiments were performed by reverse engineering two bodies of source code: the OMF's Meta-Object Facility implementation and HippoDraw. These experiments were performed by running *pilfer* against these bodies of source code to produce both XMI documents and UML diagrams (generated by GraphViz).

```
class ModelElement
{
public:
  const OMF::Set &constraints() const;
  void addConstraints(Constraint *);
  void removeConstraints(Constraint *);
  void clearConstraints(Constraints *);
  ...
private:
  OMF::Set mConstraints;
  ...
};
```

Figure 8. A snippet of the OMF's ModelElement class from the Meta-Object Facility. Dynamically typed containers such as OMF::Set impede *pilfer*'s ability to correctly identify associations.

The first test was conducted against the Open Modeling Framework's implementation of the Meta-Object Facility. To perform this test, we fed *pilfer* the source code corresponding to the implementation of that metamodel. The goal of this experiment was to recover UML diagrams that correspond to the diagrams shown in the MOF specifications [OMG'02]. Specifically, we sought to show that the UML diagrams contained the same number and types of attributes present in this specification. In this experiment, *pilfer* performs flawlessly, identifying 100% of all design-level attributes (recall) with 100% precision (no false positives). A UML diagram of a subset of the resulting model is shown in Figure 9. The listing of attributes and operations for each class

[7] See www.graphviz.org for details on GraphViz.

are identical to the diagrams given in the OMG specification. It should be noted that some of the associations and type references are missing from the model. This is an artifact of the OMF's extensive use of dynamically typed containers (such as the *OMF::Set* class shown in Figure 8). However, the model is still fairly accurate in its depiction of the Meta-Object Facility. Additionally, *pilfer*-generated models retain the UML attributes from which the associations are derived. While this is not incorrect (as per the UML specifications), it is certainly atypical for modeling purposes. We do not feel that modeling attributes for these associations detracts from the overall quality or readability of the resulting model.

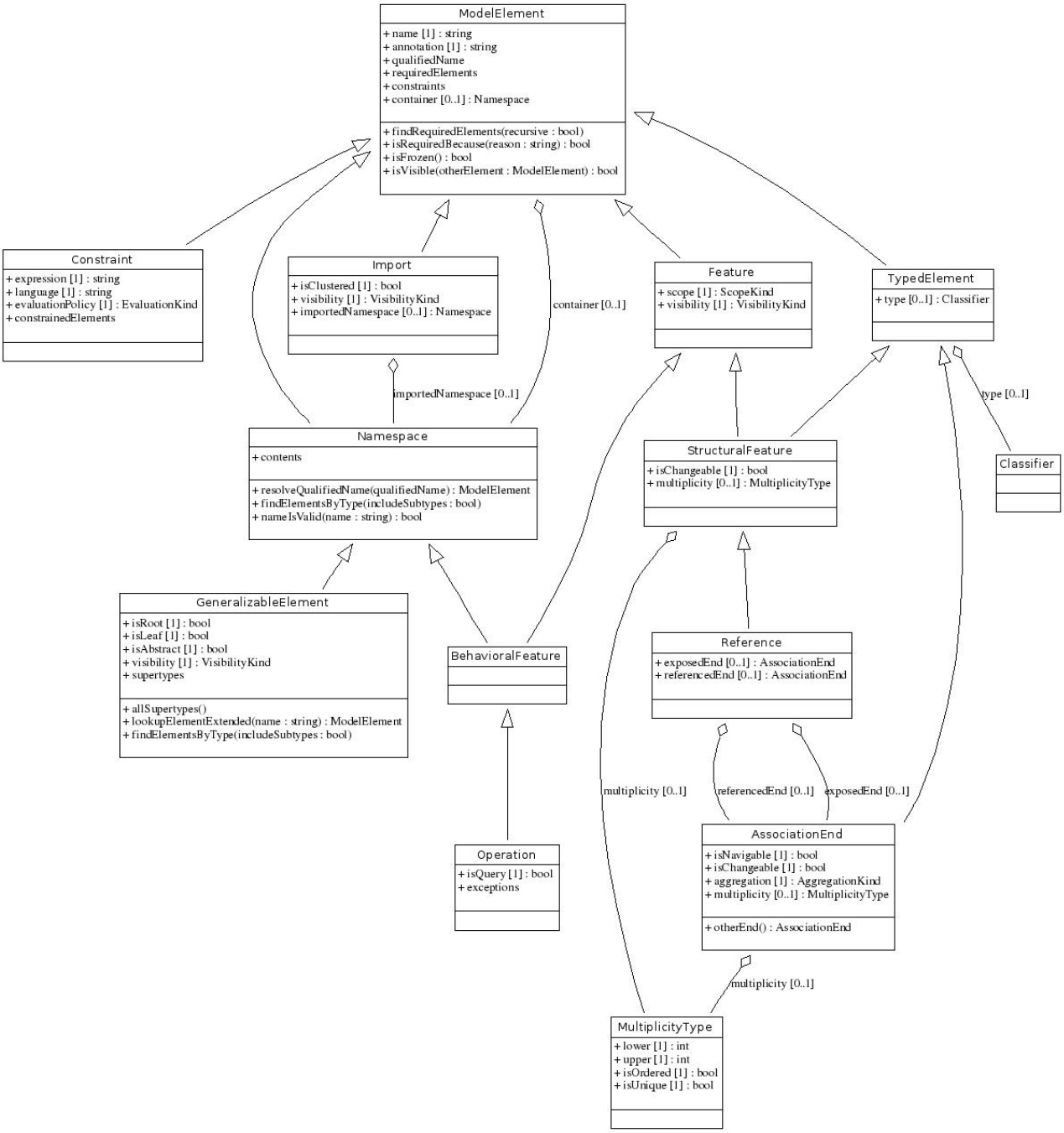

Figure 9. UML diagram of a subset of classes from the OMF implementation of the Meta-Object Facility (MOF).

In the second experiment, we ran pilfer against HippoDraw to produce an XMI model and extracted a list of UML attributes. We then compared these attributes against the member variables of classes in the HippoDraw source code to

determine whether or not the mapping produced adequate results. To do this, we examined the source code of HippoDraw and classified member variables as attributes if a set of trivial accessor and mutator functions for those member variables was found. This classification was used to produce a list of attributes that was then compared against the list derived from the XMI model. However, this process of classification was fairly naïve and only included the most trivial of accessor/mutator groupings (i.e., single line get and set methods). More complex attributes such as delegating methods were not considered as mutators. This restrictive classification was imposed in order to ensure consistency during the classification process.

In this experiment, *pilfer* recalled 96% of all attributes with a precision of 62%. In other words, the technique identified almost all of the attributes that we had manually classified as attributes, but admitted a significantly larger number of false positives than the first experiment. However, the lower precision is due to our method for manually classifying design-level attributes, which are admittedly restrictive. Inspection of these false positives conclusively shows that *pilfer* is including delegating methods in the accessor/mutator groups and producing UML attributes for the corresponding member variables.

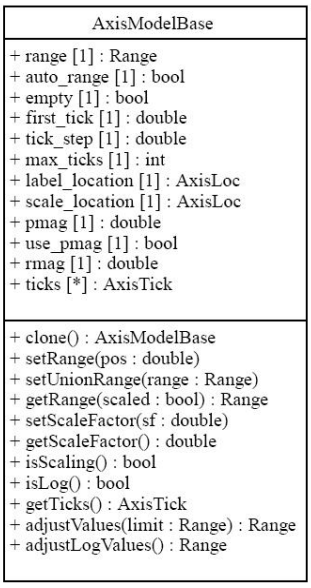

Figure 10. The UML representation of the *AxisModelBase* class produced by *pilfer*.

5. A Comparison of Results

In order to evaluate the effectiveness of our mappings and analyses, we used *pilfer* to reverse engineer HippoDraw[8] (version 1.13.1), an open source tool for information visualization. HippoDraw is a medium-sized C++ application containing about 230 classes and consists of 88 KLOC. We compared the results produced by *pilfer* against those produced by Doxygen, Visual Paradigm for UML (2005), and Microsoft Visio 2003 (used as a plugin for Visual Studio .NET 2003). We used Doxygen as a control in this comparison because it provides an accurate description of the source code. We had originally planned to conduct this experiment with two other applications supporting reverse engineering: Rational Rose C++ Edition (last version) and Umbrello 1.3. However, in both cases, the applications were unable to successfully parse the source code, making it impossible to evaluate their output. Rational Rose, in particular, failed to parse standard includes from Microsoft's implementation of the STL, and Umbrello crashed intermittently during parsing or model construction, making it an unviable candidate for this comparison. More recent versions of Rational (e.g., XDE) and Umbrello may work better. Our comparison is primarily based on the number of structural elements recovered through reverse engineering (e.g., classes, attributes, and generalizations).

Although the quantitative measures generated from these comparisons give us some insight into the completeness of the reverse engineering systems, it is difficult to ascertain the quality of the recovered models. For example, different people might have different preferences on what information is recovered or how UML semantics are inferred from the source code. This makes it nearly impossible to create a qualitative measure against a "reference model" of the application. Moreover, each modeling application expresses its own view of the source code. Those mappings from C++ to UML might not align with the mappings the user expects.

pilfer includes a number of options that allow for specific control the mappings from C++ to UML. For example, we know from experience that HippoDraw

Table 11. The number of UML elements recovered through various reverse engineering tools.

Concept Being Reverse Engineered	Reverse Engineering Tool			
	Doxygen	***Visual Paradigm***	***Microsoft Visio***	***pilfer***
Entities				
Classes	250	202	212	219
Data types	0	391	74	18
Interfaces	0	0	0	0
Attributes	889	541	626	302
Operations	7238	2809	2934	2033
Parameters	7405	5731	4198	3088
Relationships				
Association	0	92	0	77
Shared	0	0	0	65
Composite	0	0	0	12
Generalizations	201	136	190	160
Realizations	0	0	0	0

[8] See www.slac.stanford.edu/grp/ek/hippodraw/ for details onHippoDraw.

tends to use pointers to refer to single objects rather than C-arrays. As such, we configured *pilfer* to treat pointer member variables as having 0..1 multiplicity instead of 0..*.

An example of a UML class generated from *pilfer* is shown in Figure 10. The corresponding class generated by Visual Paradigm contains 18 attributes and 38 operations. The amount of information present in the Visual Paradigm class is prohibitively large and significantly reduces the readability (and therefore comprehensibility) of diagrams in which the class appears. Moreover, the Visual Paradigm parser fails to correctly parse template expressions (i.e., the *m_ticks* member, shown as *ticks* in Figure 10). Also, Visual Paradigm produces no multiplicity information about the reverse engineered attributes.

In order to further compare the results of our experiment, we have listed the number of elements recovered through the various reverse engineering tools. These results are shown in Table 11.

As mentioned, we used Doxygen as a control in the comparison. We used it to generate XML output for the classes in the system and then extracted the relevant facts using a simple XSL transform. Some of the information (such as STL classes) was removed from the resulting data in order to focus the comparison on only those classes defined within the HippoDraw application. Using Doxygen allows us to show the improvement of abstraction recovery for the various reverse engineering tools. In all other cases, scripts were written against the XMI exports of each method to extract the number of elements recovered.

Visual Paradigm was unable to parse elements in the *qt* subdirectory of HippoDraw, possibly accounting for the lower number of classes. The excessively large number of data types identified by Visual Paradigm is actually an error in their internal XMI production algorithms. They generate a new data type element for every instance of a C++ type reference. Included in this set of data types are all of the C++ primitive types (*int*, *float*, etc.). Also, the large number of parameters reflects the application's tendency to create return parameters for nullary (*void*) functions. Obviously, it is difficult to understand, in detail, how Visual Paradigm determined its rules for generating associations. It also generated a number of class-to-class dependencies, but the rationale for generating those relationships is equally unclear.

Microsoft Visio performs significantly better (especially in terms of parsing), but also includes a large number of data types. This list of data types includes the entire set of 65 primitive types from C++, C#, IDL and VisualBasic. Also, Visio models *typedefs* as UML data types rather than correctly creating import elements. Also, these measurements specifically exclude classes that Visio had rendered as "external" to the project. External classes included template instantiations and undefined classes. Unfortunately, Visio made no attempt to recover any kind of associations between elements.

Because *pilfer*, is configured differently than the other tools (i.e., its alternative method for identifying attributes), it produced a model with a significantly lower number of member variables and operations (as illustrated by Figure 10). We might also note that the data types recovered by *pilfer* are also C++ classes (giving a total of 237 C++ classes). The data types are not enumerations or *typedefs*.

For the most part, the different reverse engineering tools are in agreement about the number of classes. Had Visual Paradigm succeeded in reverse engineering the *qt* subdirectory, it would have presented numbers similar to Microsoft Visio. We might suggest the Visio presents a low number of classes because it was not configured to reverse engineer all of the subdirectories in HippoDraw; the content being reverse engineered is based on the Visual Studio project which may exclude a number of directories. A significant amount of variation also likely stems from each application's internal mechanism for handling templates instantiations – which is not always documented and much less obvious. For example, an application treating template instances as distinct classes in the system will obviously report more classes, attributes an operations.

Table 12. Approximate timing results for the execution of various reverse engineering tools against different bodies of source code.

Program	*Size (KLOC)*	*Time*
Visio	102	2m 14s
Visual Paradigm	88	>10m
g⁴re vs. FOX	125	~20m
g⁴re vs. Jikes	70	~11m
pilfer	102	1m 42s

In general, support for recovering associations, specifically aggregation semantics, in the studied tools is insufficient. It appears the tools would rather avoid making decisions about association semantics than possibly make a wrong decision. Among the systems studied only *pilfer* provides any significant information about associations between classes.

In terms of performance, *pilfer* does quite well with respect to the work of others who reported performance information in the literature and those we could measure. While this is certainly not an exact comparison between these tools, it is given to provide a general sense of the run time performance rather than a normative benchmark. In addition to timing comparisons against the applications above, we also compare *pilfer* against *g⁴re*, [Kraft'05] an interoperable reverse engineering tool based on a GCC intermediate representation (translation unit files). To obtain our performance

numbers, we ran both Microsoft Visio and Visual Paradigm on an Intel P4 1.7 GHz machine with 1 GB of RAM. *pilfer* was run on an Intel P4 2.8 GHz machine with 1 GB of RAM, running Linux. g*4re* was reported to run on an Athlon64 3000+ with 1 GB of RAM running Linux. The timing results are shown in Table 12.

Unfortunately, the comparisons are skewed for a number of reasons – especially the use of different platforms, different operating systems, and different implementation languages. Moreover, not all times are reported for the same inputs. For example, the Visual Paradigm was unavailable (due to licensing) at the time of this performance test, so we could not run it against the same version of HippoDraw (i.e., 1.16.0) that Microsoft Visio and *pilfer* were tested against. Also, the recorded time is only an approximation because early experiments did not record timing performance accurately. The *g4re* application was tested against two systems. FOX is a C++ GUI toolkit and Jikes is an experimental Java compiler.

Microsoft Visio and *pilfer* perform at roughly the same level, and we can imagine that Visio would perform better if run on a similarly equipped system. Visual Paradigm seemed to take the longest – perhaps because of its Java roots or because of memory usage problems. Despite the slower times of *g4re*, its analysis times are actually quite good. The vast majority of the time spent by this application is in processing the output of GCC's GENERIC module. It produces XML files containing the abstract syntax graph (ASG) of each compiled translation unit, resulting in an enormous amount of data. Contrast this with the output size of srcML which is typically less than five times the original data size, resulting in significantly less XML processing.

6. Related Work

The prevailing method of integrating modeling and reverse engineering tools is to build reverse engineering parsers and analyses into existing UML modeling applications. Examples include Rational Rose, Together, Umbrello, Visual Paradigm, and ArgoUML. However, IDE's are beginning to realize the importance of providing a visual medium for source code and have begun to include UML modeling functionality. Both Microsoft's Visual Studio 2005 and Apple's XCode2 both support the ability to model classes in UML-like diagrams.

It is a matter of fact that most advances in reverse engineering research have little impact on industrial reverse engineering tools despite the fact that research technologies are addressing many of the shortcomings of industry applications. For example, numerous approaches have been described for reverse engineering associations and related information from Java. [Barowski'02] proposes a technique to extract dependency information from Java class (bytecode) files. The approach is used to detect inheritance, realization, and associations between classes. However, this definition of dependency is overly general and does not correspond to more specific relationships of UML. Moreover, the approach is not capable of expressing multiplicity or aggregation semantics of the recovered associations. [Guéhéneuc'04] provides a much stronger definition of associations at both the design and implementation level for both aggregate and composite associations. An algorithm for the detection of these associations is derived from a formal analysis of their properties and applied to a number of Java systems, performing quite well (96% recall and 75% precision). The approach described in [Jackson'99] applies heuristics to static and semantic analysis of Java class files. This work is very much like our own approach as it constructs object models (class models) using lightweight syntactic and semantic analysis. However, their implementation *Womble*, only operates on Java bytecode whereas *pilfer*, operates on C++ source code.

In [Gogolla'00], UML associations, multiplicities and aggregation semantics are inferred by examining the source code. Additionally, this approach defines one of the few techniques for finding bi-directional associations. [Kollman'01] uses both static and dynamic analysis to accomplish many of the same goals. Yet another approach to identifying the multiplicity of associations is given in [Keschenau'04]. To our knowledge (possibly because of poor documentation), none of these approaches have been integrated into any of the prevalent reverse engineering tools.

Many of the analysis methods in the works listed above are not applicable to C++, especially those operating on bytecode. Unfortunately, related research on reverse engineering methods for C++ is significantly less. In [Tonella'01], type information is reverse engineered from C++ by examining usage patterns of so-called weakly-typed containers. This approach can be used to identify which classes are used in containers if the container collects instances of abstract base classes such as the ubiquitous *Object* class. Although [Tonella'02, '03] focus more on recovering behavioral aspects of a program, they illustrate additional techniques for reverse engineering information from C++. In these approaches, an object flow graph is coupled with other forms of analysis to produce object and interaction diagrams, respectively. [Matzko'02] takes an approach similar to ours – the implementation of a C++ reverse engineering environment. However, this work focused on the modeling of C++ syntax in UML rather than attempting to recover the design abstractions involved. Moreover, little is said about the rules used to recover multiplicities or aggregation semantics.

Formal concept analysis is being applied to a number of different areas in software engineering many of which are surveyed in [Tilley'03]. In legacy systems, formal concept analysis has been applied to preprocessor configurations in order to better understand and help maintain complicated portability configurations [Krone'94, Snelting'98]. Concept analysis is frequently used to construct object oriented models from legacy or procedural code. Examples include [Canfora'99, Sahraoui'97, Siff'99, van Deursen'99]. It has also been used to analyze the coupling and cohesion between modules in [Lindig'97].

Another popular application of concept analysis is to use it to reorganize or analyze class hierarchies in object-oriented software [Godin'93, Godin'98, Snelting'00, Snelting'98]. In [Tonella'99], concept lattices are used to detect design patterns by analyzing inheritance relationships. Additionally, concept lattices are used as a visualization technique for information extracted from source code in [Cole'03].

One innovative use of concept analysis has been in the area of program comprehension via feature location [Eisenbarth'01, '03]. In this research, concept analysis is used in conjunction with both dynamic and static analysis. Specifically, dynamic analysis is used to generate sets of related scenarios and subprograms. Concept analysis is used to identify features by relating which subprograms are invoked in each scenario. Static (call graph) analysis is then used to augment the information in the concept lattice, providing a detailed visualization the feature's implementation.

What distinguishes this work from other applications of concept analysis is primarily the scope to which the analysis is applied. These applications typically consider the entire scope of a program – sometimes generating an enormous amount of information that resists further analysis, much less visualization [Anquetil'00]. In our approach, we consider object/attribute on a class-by-class basis rather than the entire system as a whole. One application similar to ours is [Dekel'03], in which concept lattices are constructed for individual Java classes to support program comprehension. Our approach varies in that the resulting lattices are used to determine additional qualities of a class rather than producing visualization material.

7. Conclusions and Future Work

In this paper we have discussed the inconsistency of reverse engineering tools due to the semantic gap between UML and C++ and the non-disclosure policy of those tools. In an effort to bridge the gap between the two languages and to provide a platform for common modeling problems, we have detailed a set of mappings from C++ to UML class models. These heuristic mappings are based primarily on easily accessible syntactic and semantic information in the program. These mappings are intended to recover design-level UML class models from source code, supporting program comprehension.

To validate the applicability and accuracy of these mappings, we have implemented them in our own reverse engineering application *pilfer*. When used to reverse engineer HippoDraw, it produced UML models comparable to those produced by other reverse engineering tools. However, the inclusion of domain knowledge allowed *pilfer* to produce models that reflected the abstract design rather than the simply recreating the implementation-level structures of the program. This is immediately obvious in a) the identification of abstract attributes and b) the presence of appropriate aggregation semantics in associations.

In the future we plan to expand our investigation of common ambiguities and inconsistencies between reverse engineering tools and define similar mappings for their resolution. Moreover, we plan to refine the architecture of *pilfer* to allow even more user customization, scriptability, and interoperability. Allowing users to develop and integrate more analysis technologies is a priority for creating a more complete and accurate reverse engineering environment. Although *pilfer* is already capable of producing standards-compliant XMI for model interchange, and *dot* files for visualization of UML diagrams and concept lattices, we envision a broader range of exported formats such that we can interoperate with a broad range of applications in the reverse engineering community. Efforts are being undertaken by the research community to define such formats [Holt'00, Malton'05]. Moreover, interoperability is being increasingly discussed as a requirement for reverse engineering tools as a whole (e.g., [Kraft'05]). To support these broader goals of the community, we envision using *pilfer* to export, say, GXL models of call graphs, object relation diagrams and other such artifacts, allowing other applications to analyze or visualize the results.

8. Acknowledgments

We would like to thank the reviewers for their helpful and detailed comments in revising this paper. This work was supported in part by a grant from the United States National Science Foundation (C-CR 02-04175).

9. References

[Anquetil'00] Anquetil, N., (2000), "A Comparison of Graphs of Concept for Reverse Engineering", in Proceedings of 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 10-11, pp. 231 - 240.

[Barowski'02] Barowski, L. A. and Cross, J. H., (2002), "Extraction and Use of Class Dependency Information in Java", in Proceedings of Ninth Working Conference on Reverse Engineering (WCRE'02), Richmond, Virginia, Oct 29-Nov 1, pp. 309-318.

[Baxter'04] Baxter, I. D., Pidgeon, C., and Mehlich, M., (2004), "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 - 28, pp. 625-634.

[Canfora'99] Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G. A., (1999), "A Case Study of Applying an Eclectic Approach to Identify Objects in Code", in Proceedings of 7th International Workshop on Program Comprehension (IWPC'99), Pittsburgh, Pennsylvania, pp. 136-143.

[Chikofsky'90] Chikofsky, E. J. and Cross, J. H., (1990), "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, vol. 7, no. 1, January, pp. 13-17.

[Cole'03] Cole, (2003), "Conceptual Analysis of Software Structure", in Proceedings of 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03), San Francisco Bay, California, Jul 1-3, pp. 726-733.

[Collard'03] Collard, M. L., Kagdi, H. H., and Maletic, J. I., (2003), "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11, pp. 134-143.

[Collard'02] Collard, M. L., Maletic, J. I., and Marcus, A., (2002), "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9, pp. 34-41.

[Dekel'03] Dekel, U. and Gil, Y., (2003), "Revealing Class Structure with Concept Lattices", in Proceedings of 10th Working Conference on Reverse Engineering (WCRE'03), Victoria, Canada, Nov 13-16, pp. 353-365.

[Eisenbarth'01] Eisenbarth, T., Koschke, R., and Simon, D., (2001), "Aiding Program Comprehension by Static and Dynamic Feature Analysis", in Proceedings of International Conference on Software Maintenance (ICSM01), Florence, Italy, November 7-9, pp. 602-611.

[Eisenbarth'03] Eisenbarth, T., Koschke, R., and Simon, D., (2003), "Locating Features in Source Code", IEEE Transactions on Software Engineering, vol. 29, no. 3, March, pp. 210 - 224.

[Fowler'00] Fowler, M.,(2000),UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language, Addison-Wesley.

[Godin'93] Godin, R., Mili, H., Mineau, G., Missaoui, R., Arfi, A., and Chau, T.-T., (1993), "Building and Maintaining Analysis-Level Class Hierarchies using Galois Lattices", in Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Application, Washington, D.C., Sep 26 - Oct 1, pp. 394-410.

[Godin'98] Godin, R., Mineau, G., Missaoui, R., Arfi, A., and Chau, T.-T., (1998), "Design of Class Hierarchies based on Concept (Galois) Lattices", International Journal of Knowledge Engineering and Software Engineering, vol. 5, no. 1, 1998, pp. 119-142.

[Gogolla'00] Gogolla, M. and Kollman, R., (2000), "Re-Documentation of Java with UML Class Diagrams", in Proceedings of 7th Reengineering Forum, Reengineering Week 2000, Zurich, Switzerland, Feb 29 - Mar 3, pp. 41-48.

[Guéhéneuc'04] Guéhéneuc, Y.-G. and Albin-Amiot, H., (2004), "Recovering Binary Class Relationships: Putting Icing on the UML Cake", in Proceedings of 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), Vancouver, Canada, Oct 24-28, pp. 301-314.

[Holt'00] Holt, R. C., Winter, A., and Schürr, A., (2000), "GXL: Toward a Standard Exchange Format", in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, November, 23 - 25, pp. 162-171.

[Jackson'99] Jackson, D. and Waingold, A., (1999), "Lightweight Extraction of Object Models from Bytecode", in Proceedings of 21st International Conference on Software Engineering (ICSE'99), Los Angeles, California, May 16-22, pp. 194-202.

[Jiang'03] Jiang, J. and Systa, T., (2003), "Exploring Differences in Exchange Formats - Tool Support and Case Studies", in Proceedings of Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, March 26-28, pp. 389-398.

[Keschenau'04] Keschenau, M., (2004), "Student Research Competition: Reverse Engineering of UML Specifications from Java Programs", in Proceedings of Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, Vancouver, Canada, Oct 24-28, pp. 326-327.

[Kollman'01] Kollman, R. and Gogolla, M., (2001), "Application of UML Assciations and Their Adornments in Design Recovery", in Proceedings of Eigth Working Conference on Reverse Engineering (WCRE'01), Stutgart, Germany, Oct 2-5, pp. 81-92.

[Kollman'02] Kollman, R., Selonen, P., Stroulia, E., and Zündorf, A., (2002), "A Study in the Current State of the Art in Tool-supported UML-based Static Reverse Engineering", in Proceedings of Ninth Working Conference on Reverse Engineering (WCRE'02), Richmond, Virginia, Oct 29-Nov 1, pp. 22-34.

[Kraft'05] Kraft, N. A., Malloy, B. A., and Power, J. F., (2005), "Toward an Infrastructure to Support Interoperability in Reverse Engineering", in Proceedings of 12th Working Conference on Reverse Engineering (WCRE ' 05), Pittsburgh, PA, Nov 7-11, pp. 196-205.

[Krone'94] Krone, M. and Snelting, G., (1994), "On the Inference of Configuration Structures from Source Code", in Proceedings of 16th International Conference on Software Engineering (ICSE'94), Sorento, Italy, May 16-21, pp. 49-57.

[Leigh'99] Leigh, J., Johnson, A. E., Brown, M., Sandin, D., and Defanti, T. A., (1999), "Visualization in Teleimmersive Environments", IEEE Computer, vol. 32, no. 12, December, pp. 66-73.

[Lindig'97] Lindig, C. and Snelting, G., (1997), "Assessing modular structure of legacy code based on mathematical concept analysis", in Proceedings of International Conference on Software Engineering (ICSE'97), Boston, MA, May 17-23, pp. 349 - 359.

[Maletic'02] Maletic, J. I., Collard, M. L., and Marcus, A., (2002), "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29, pp. 289-292.

[Malton'05] Malton, A. J. and Holt, R. C., (2005), "Boxology of NBA and TA: A Basis for Understanding Software Architecture", in Proceedings of 12th Working Conference on Reverse Engineering (WCRE ' 05), Pittsburgh, PA, Nov 7-11, 2005, pp. 187-195.

[Matzko'02] Matzko, S., Clarke, P. J., Gibbs, T. H., Malloy, B. A., Power, J. F., and Monahan, R., (2002), "Reveal: A Tool to Reverse Engineer Class Diagrams", in Proceedings of 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, Sydney, Australia, 2002, pp. 13-21.

[OMG'02] OMG, (2002), "Meta Object Facility (MOF), 1.4": http://www.omg.org.

[OMG'03] OMG, (2003), "Unified Modeling Language, 1.5": http://www.omg.org.

[Sahraoui'97] Sahraoui, H. A., Melo, W., Lounis, H., and Dumont, F., (1997), "Applying concept formation methods to object identification in procedural code", in Proceedings of International Conference on Automated Software Engineering (ASE '97), Lake Tahoe, CA, November 02 - 05.

[Siff'99] Siff, M. and Reps, T. W., (1999), "Identifying Modules via Concept Analysis", IEEE Transactions on Software Engineering, vol. 25, no. 6, November/December, pp. 749-768.

[Snelting'00] Snelting, G., (2000), "Software Reengineering based on Concept Lattices", in Proceedings of 4th European Conference on Software Maintenance and Reuse (CSMR'00), Zurich, Switzerland, Feb 29 - Mar 03, pp. 3-12.

[Snelting'98] Snelting, G. and Tip, F., (1998), "Reengineering Class Hierarchies Using Concept Analysis", in Proceedings of SIGSOFT, pp. 99-110.

[Storey'97] Storey, M.-A. D., Wong, K., and Müller, H. A., (1997), "Rigi: a visualization environment for reverse engineering", in Proceedings of IEEE International Conference on Software Engineering (ICSE'97), Boston, MA, May 17 - 23, pp. 606-607.

[Sutton'05] Sutton, A., (2005), Accurately Reverse Engineering UML Class Models from C++, Kent State University, Kent, Ohio, Masters Thesis.

[Sutton'05] Sutton, A. and Maletic, J. I., (2005), "Mappings for Accurately Reverse Engineering UML Class Models from C++", in Proceedings of 12th Working Conference on Reverse Engineering (WCRE ' 05), Pittsburgh, PA, Nov 7-11, pp. 175-184.

[Tilley'03] Tilley, T., Cole, R., Becker, P., and Eklund, P., (2003), "A Survey of Formal Concept Analysis Support for Software Engineering Activities", in Proceedings of 1st International Conference on Formal Concept Analysis (ICFCA'03), Darmstadt, Germany, Feb 27 - Mar 1.

[Tonella'99] Tonella, P. and Antoniol, G., (1999), "Object-Oriented Design Pattern Inference", in Proceedings of 3rd European Conference on Software Maintenance and Reuse (CSMR'99), St. Agnes, Amsterdam, Mar 3-5, pp. 230-240.

[Tonella'01] Tonella, P. and Potrich, A., (2001), "Reverse Engineering of the UML Class Diagram from C++ Code in the Presence of Weakly Typed Containers", in Proceedings of International Conference on Software Maintenance (ICSM'01), Florence, Italy, Nov 6-10, pp. 376-385.

[Tonella'02] Tonella, P. and Potrich, A., (2002), "Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram", in Proceedings of International Conference on Software Maintenance (ICSM'02), Montreal, Canada, Oct 3-6, pp. 54-63.

[Tonella'03] Tonella, P. and Potrich, A., (2003), "Reverse Engineering of the Interaction Diagrams from C++ Code", in Proceedings of International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands, Sep 22-26, pp. 159-168.

[van Deursen'99] van Deursen, A. and Kuipers, T., (1999), "Identifying Objects using Cluster and Concept Analysis", in Proceedings of 21st IEEE International Conference on Software Engineering (ICSE'99), Los Angeles, CA, May 16-22, pp. 246-255.

[Wong'95] Wong, K., Tilley, S. R., Muller, H., and Storey, M.-A. D., (1995), "Structural Redocumentation: A Case Study", IEEE Software, vol. 12, no. 1, January, pp. 46-54.