Mappings for Accurately Reverse Engineering UML Class Models from C++

Andrew Sutton, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{asutton, jmaletic}@cs.kent.edu

Abstract

The paper introduces a number of mapping rules for reverse engineering UML class models from C++ source code. The mappings 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. An application implementing these heuristics is used to reverse engineer a moderately sized open source, C++ application, and the resultant class model is compared against those produced by other UML reverse engineering applications. A comparison shows that these presented mapping rules effectively produce meaningful, semantically accurate UML models.

1. Introduction

Many software engineers would greatly appreciate a tool that accurately reverse engineers UML class models from (real) C++ source code. Unfortunately, the tools most often used in practice perform fairly poorly at this task. A case study of reverse engineering applications [10] found 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 becomes a problem when recovered software models fail to accurately represent the abstract program semantics required for high-level program comprehension.

Although this study [10] concludes that these applications provide reliable functionality, the resulting models are anything but consistent. For example, Microsoft Visio is incapable of reverse engineering associations and Visual Paradigm creates dependencies when associations are appropriate. In the case of the Rational Rose C++ Modeler it only creates shared aggregation associations (when it is able to parse the code). The primary reason for this inconsistency 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 applications implement fairly closed systems, providing little information about the how the UML models are created from C++, leaving 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 applications 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. 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 bring together what has been presented in the literature dealing with this topic.

These mappings are implemented in a reverse engineering tool, *pilfer*, which is used to test the relevance of the defined mappings by reverse engineering a moderately-sized C++ application, namely HippoDraw. The model produced by *pilfer* is compared against those produced by other applications in order to validate the accuracy and completeness of the defined mappings.

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. In section 4, we present a comparison of models generated by *pilfer* and other reverse engineering applications. Section 5 describes work related to this topic and section 6 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 [3]. 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.

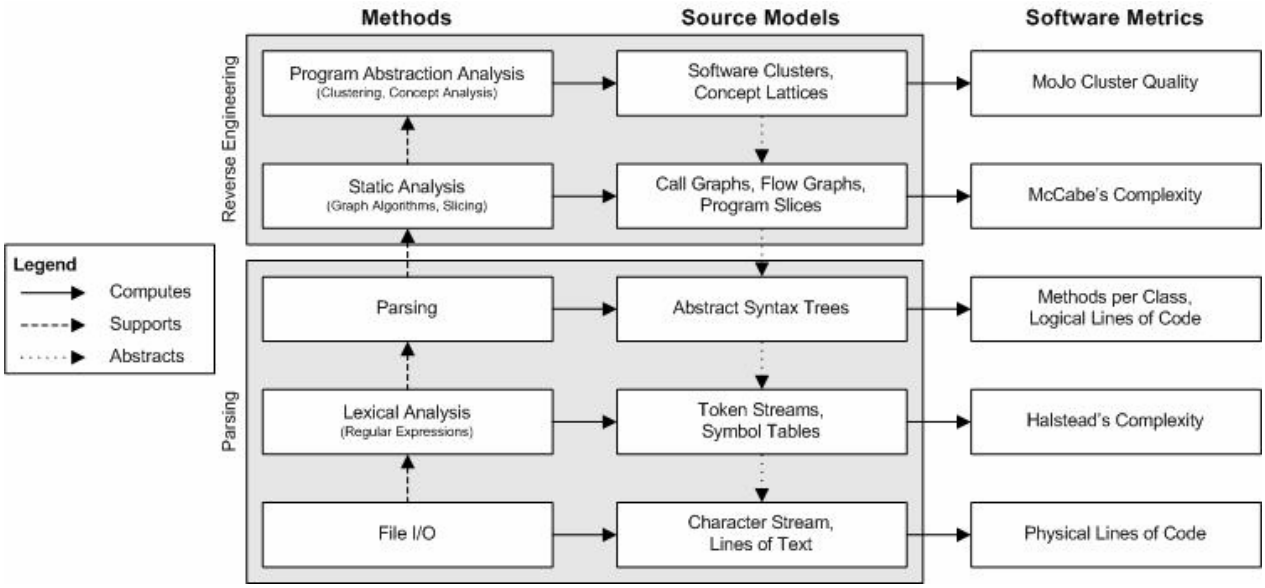

Figure 1. A reverse engineering stack produces increasingly abstract source models based on layered processes and analyses. Metrics are often computed for source models at different levels of abstraction.

This technology stack is motivated in part by the *Rigi* reverse engineering environment [12, 16] and the DMS program analysis system [2]. 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 (e.g., lexical analysis requires file I/O and supports parsing). 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, an abstraction has a number of potential metrics that can be computed from the resultant source model.

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 typically allow users to embed their own specific domain knowledge into the reverse engineering processes. Moreover, with a limited set of analysis methods, these applications 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 reference tool 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 to be reverse engineered from source code. These concepts are representative of current deficiencies in reverse engineering applications as noted in [10] and through our practical experience with a number of reverse engineering applications. 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.

For each task in Table 1, we have listed several observations. They are as follows: the *required analysis* column describes the minimum required analysis technique to accomplish each task. Simple parsing implies that the results could be obtained through the AST of the program. Semantic analysis means that the

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*
	Simple Parsing	*Semantic Analysis*	*Static Analysis*			
Entities						
Classes	X				None	
Identify interfaces		X		X	Med	Distinguish interfaces from classes
Identify data types		X		X	Med	Distinguish data types from classes
Attributes	X				None	
Use the property idiom			X	X	High	Identify attributes from accessors/mutators
Identify read-only attributes		X		X	Low	Attributes with only accessors
Type resolution		X		X	Med	Correctly resolve type references
Multiplicity			X	X	Med	Identify multiplicity of attributes
Relationships						
Association		X			High	Which attributes constitute associations
Aggregation semantics			X	X	High	Determine object lifetime
Realization		X			Med	Classes implementing interfaces

application must rely upon specific C++ semantics in order to perform the task. 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. The *degree of ambiguity* describes how consistently different applications perform the task. Low ambiguity implies that applications produce mostly consistent models, whereas high ambiguity tasks result in widely varying results. The *description* column provides additional details for the reverse engineering task.

Note that we do not include methods in this table because current reverse engineering tools typically have little difficulty identifying the methods of a class. Instead we are interested in *which* methods are actually produced in the UML model. Most reverse engineering applications identify constructors, destructors, and overloaded operations in the reverse engineered classes. However, we see these as implementation details that only serve to clutter the resultant model. In this case, we recommended allowing the user to decide the relevance of those methods.

3. Mappings for Reverse Engineering

In this section, we define mappings for UML reverse engineering tasks with a degree 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 straight forward. 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.

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 don't participate in associations). These restrictions have been relaxed in newer versions.

```
// This is a true C++ interface
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.

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.

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 author is relying on the compiler to supply defaults for these methods.

Automating this mapping might lead to cases where classes are mis-identified 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).

```
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.

3.2. Attributes

Typical reverse engineering applications correlate UML attributes with a class's member variables. However, in Model-Driven Architecture (MDA) forward engineering approaches, attributes are used quite 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 applications 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 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.

```
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.

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 affects if behaviors of a class are mis-modeled as 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 declarations are unclear.

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"

3.2.1. Type Resolution. 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.

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

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

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

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 represent this transitive property and which template parameters correspond to the appropriate type information.

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 we can identify 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.

3.3. Identifying Associations

Although UML associations are most often used to represent "has-a" relationships, they are sometimes employed to model semantic connections between two different classes. C++ does not provide a syntactic concept for modeling these semantic links. However, it is fairly easy to see 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++ class types and declarative type information. We define a C++ association as a member variable with a type reference to another 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.

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]`	n..n
`Foo *[n]`	0..n
`Foo **`	0..*
`Foo &`	1..1
`list<Foo>`	0..*
`auto_ptr<Foo *>`	0..1

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*.

```
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.

3.3.1. Aggregation. Aggregation defines object lifetime semantics for instances contained by a property. UML defines three types of containment semantics for attributes participating in associations: *none*, *shared*, and *composition*. The latest versions of UML associate aggregation directly with an attribute (property) rather than the association (ends). The only types of aggregation that can be derived from the C++ grammar are shared aggregation and composition. The *none* variety of aggregation 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 6 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.

Table 6. 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]	Shared
`Foo &`[2]	Shared
`Foo[]`	Composite
`Foo *[]`	Shared
`auto_ptr<>`[3]	Shared
`boost::scoped_ptr<>`	Composite
`boost::shared_ptr<>`	Shared

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 6. Specific domain knowledge of these classes must be used to deduce aggregation semantics.

3.4. Generalization and Realization

In UML, inheritance is represented by generalization relationships. 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 class 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 should be modeled as well.

4. A Comparison of Results

In order to evaluate the effectiveness of our mappings, we have implemented an application, *pilfer*, that reverse engineers C++ source code into UML models. The application is written in the Python programming language and leverages srcML [4] for a parsing platform. The Open Modeling Framework (OMF), an implementation of the UML metamodel that we have built, is used to construct the UML models. The *pilfer* application implements all of the mappings discussed in Section 3. Additionally, we have implemented sophisticated program analysis employing concept lattices to automatically deduce abstract attributes from a class's member variables in order to provide more abstract design information and make the resulting diagrams more readable.

We used *pilfer* to reverse engineer HippoDraw (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 resulting produced by *pilfer* against those produced by Doxygen, Visual Paradigm for UML, and Microsoft Visual Studio and Visio. We primarily used Doxygen as a control in this comparison because it provides a fairly accurate picture of the code. We had originally planned to conduct this experiment with two other applications supporting reverse engineering: Rational Rose C++ Edition and Umbrello. However, in both cases, the applications were unable to successfully parse the source code, making it impossible to evaluate their output. However, more recent versions of Rational 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 (or that of its developers) 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 over how the system maps from C++ to UML. For example, we know from experience that HippoDraw uses STL container classes to implement collections rather than C-vectors. As such, we configured *pilfer* to treat pointer member variables as having 0..1 multiplicity instead of 0..*.

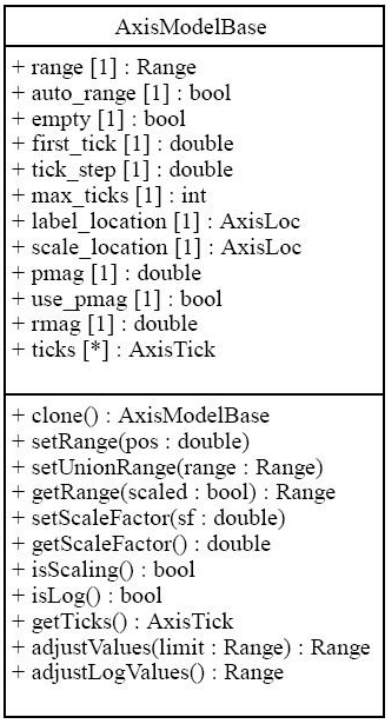

Figure 6. The UML representation of the *AxisModelBase* class produced by *pilfer*.

An example of a UML class generated from *pilfer* is shown in Figure 6. 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 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 6). 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 7.

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 only on those classes defined within the HippoDraw application. Using Doxygen allows us to show the improvement of abstraction recovery for the various reverse engineering applications. In all other cases, scripts were written against the XMI exports of each method to extract the number of elements recovered.

Table 7. The number of UML elements recovered through various reverse engineering methods.

Concept Being Reverse Engineered	*Reverse Engineering Method*			
	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

Visual Paradigm was unable to parse elements in the *qt* subdirectory, 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 redefinitions. 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 applications (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 6). 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 methods 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.

In general, support for recovering associations, specifically aggregation semantics, in the studied applications is insufficient. It appears the applications 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.

5. Related Work

The prevailing method of integrating modeling and reverse engineering applications 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. However, most of these applications provide little flexibility for users wanting to alter, extend, or integrate additional analyses into the systems.

It is a matter of fact that most advances in reverse engineering research have little impact on industrial reverse engineering applications 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. [1] 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 that UML provides. Moreover, the approach is not capable of expressing multiplicity or aggregation semantics of the recovered associations. [6] 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 [7] applies heuristics to static and semantic analysis of Java class files. This work is very much like our own approach. In [5], 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. [9] 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 [8]. To our knowledge (possibly because of poor documentation), none of these approaches have been integrated into any of the prevalent reverse engineering applications.

Because all of these techniques focus on Java, many of the analyses and heuristics do not translate directly into C++. Unfortunately, related research in C++ is significantly less. In [13], 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. Although [14, 15] 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.

[11] 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.

6. Conclusions and Future Work

In this paper we have discussed the inconsistency of reverse engineering applications due to the semantic gap between UML and C++ and the non-disclosure policy of those applications. In an effort to bridge the gap between the two languages and to provide a platform for common modeling problems, we have defined 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.

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 applications. However, the inclusion of domain knowledge allowed *pilfer* to produce models that reflected the abstract design rather than the simply recreating the structure 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 applications and define similar mappings for their resolution. Moreover, we plan to refine the architecture of *pilfer* to allow even more user customization. Allowing users to develop and integrate more analysis technologies is a priority for creating a more complete and accurate reverse engineering environment.

This work was supported in part by a grant from the National Science Foundation (C-CR 02-04175).

7. References

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

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

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

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

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

[6] Guéhéneuc, Y.-G. and Albin-Amiot, H., "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 2004, pp. 301-314.

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

[8] Keschenau, M., "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 2004, pp. 326-327.

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

[10] Kollman, R., Selonen, P., Stroulia, E., and Zündorf, A., "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, Oct 29-Nov 1 2002, pp. 22-34.

[11] Matzko, S., Clarke, P. J., Gibbs, T. H., Malloy, B. A., Power, J. F., and Monahan, R., "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, 2002 2002, pp. 13-21.

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

[13] Tonella, P. and Potrich, A., "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 2001, pp. 376-385.

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

[15] Tonella, P. and Potrich, A., "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 2003, pp. 159-168.

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