Exchange Of Software Representations Among Reverse Engineering Tools

Dean Jin
jin@cs.queensu.ca

December 4, 2001
External Technical Report
ISSN-0836-0227-
2001-454

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Abstract

This paper examines the exchange of software representations among reverse engineering tools. Background information on maintenance related activities and their importance in the software development lifecycle are outlined. An overview of tool support for software maintenance demonstrates the need for a standardized means for facilitating the exchange of information among reverse engineering tools. A variety of techniques for exchanging software representations are examined with respect to their relative advantages and disadvantages. The characteristics of a number of software exchange formats are summarized among various taxonomies. Four different types of exchange are characterized. Each is evaluated on how it satisfies the requirements for a standard exchange format. The paper concludes with a look at the direction research efforts are taking towards enabling the exchange of software representations among reverse engineering tools in the near future.

Contents

List of Figures

List of Tables

1 Introduction

Much of the focus within the field of software engineering over the past 50 years has been on the creation of new software products. As a result, many complex, large-scale software systems have been constructed. The past decade has seen an increased awareness of the challenges involved in maintaining these systems. Software maintenance involves many tasks of which *reverse engineering* [CC 90] is perhaps the most important. It involves analyzing a system to determine how it is constructed, resulting in the creation of representations that aid in system comprehension. Various tools designed to assist maintainers in carrying out reverse engineering have been created. Most of these tools are effective for specific types of analysis but are weak in other areas. No single tool exists that provides all the functionality and flexibility that most software maintainers need. For this reason, research attention has been focused on getting reverse engineering tools to interoperate with each other.

The interchange of software representations is widely considered the key enabler of interoperability among reverse engineering tools. Recently, a debate on how to exchange software representations among reverse engineering tools has been pursued more rigorously. A number of conferences and research groups from the reverse engineering and graph drawing community have stimulated discussion and brought about some progress in this area.

This paper will examine the exchange of software representations among reverse engineering tools. First we provide background information on maintenance related activities and outline their importance in the software development lifecycle. An overview of tool support for software maintenance will demonstrate the need for a standardized means for facilitating the exchange of information among reverse engineering tools. A variety of techniques for exchanging software representations will be outlined and examined with respect to their relative advantages and

disadvantages. The characteristics of a number of software exchange formats will be summarized among various taxonomies. We go on to characterize various types of exchange and evaluate them on how they satisfy the requirements for a standard exchange format. One format in particular has been widely accepted among researchers for exchanging software representations. The motivation behind this recognition will be outlined. We conclude with a look at the direction research efforts are taking towards enabling the exchange of software representations among reverse engineering tools in the near future.

1.1 Focus of Discussion

Software representation, as a subtopic within the broad field of data representation, has applications in both forward and reverse engineering disciplines. In this paper our focus is specifically directed towards the interchange of software representation among reverse engineering tools.

The use of graphical means to design and drive the development of software is a well-established practice in forward software engineering. In the last decade the popularity of object-oriented analysis and design methods has led to the creation of tools that support the creation and modification of *Unified Modeling Language (UML)* [OMG 01a] diagrams and other object-based software representations. In a forward engineering process, software is initially represented as highly abstracted architectural or design artifacts. Subsequent development steps see the software represented at lower levels of abstractive detail concluding with abstraction-free source code.

Reverse engineering starts with source code artifacts and, through manual or automated techniques, proceeds to increase the level of abstraction in the software representation as progression is made toward the recovery of design and architectural elements. At the outset, reverse engineering requires an abstraction-free means for representing source code. Forward

engineering, with its focus on abstraction, does not offer a representational semantics that is compatible with reverse engineering. It is not surprising that Demeyer, Ducasse and Tichelaar [DDT 99] argue that UML lacks "concepts that are necessary in order to adequately model source-code".

Needless to say, interchange of software representations among forward engineering tools has been achieved in a limited fashion. Computer-Aided Software Engineering (CASE) tools typically provide a cohesive means (although often a proprietary, internalized format) for representing software through different phases of a structured development process. More recently, a number of formats for exchanging UML and object-based models in the context of forward software engineering have been proposed including Microsoft's *XIF* [Mic 99], IBM and the Object Management Group's *XMI* [IBM 00, OMG 98], Rational Software's *UML-Compliant Interchange Format* [RAT 97], Rivard's *UML-Xchange* [Riv 00b] and Suzuki and Yamamoto's *UXF* [SY 98a, SY 98b]. The Object Management Group is currently seeking input on a diagram interchange format for the second version of UML [OMG 01b].

It is notable that some graph-based exchange formats exist within the graph drawing community that are capable of representing software for reverse engineering purposes. Some examples are *daVinci* [daV 98a, daV98b, FW 94], *dot* [GN 99, KN 96], *GEL* [Kam 94], *GDL* [San 95], *GML* [GML 01, Him 97], *GraphEd* [Him 94, Him 89], *GraphXML* [HM 00a, HM 00b], *GRL* [PT 90], the *Graph Layout Toolkit* format [TSC 98] and *XGMML* [Cov 00]. In general these formats exist as a means for communicating graphs among various tools for visualization and graph transformation analysis. We have restricted our discussion of exchange formats in this paper to those that have actually been used for representing software in the context of reverse engineering.

2 Maintenance

Maintenance involves the modification of a software system after it has been delivered. According to Corbi [Cor 89], maintenance is performed for one of four possible reasons:

1. **Correction**. Problems found after the software has been delivered need to be fixed.
2. **Required Changes**. External factors force a change in existing software to accommodate new functionality.
3. **Enhancement**. Existing software is changed to provide new features or functionality for users.
4. **Pre-emptive Changes**. The system is upgraded to facilitate future maintenance ease.

In Figure 1 the incidences of each type of maintenance is shown. It is notable that only 20% of all maintenance activity involves the correction of defects in existing software. The remaining 80% (i.e. enhancement, required changes and pre-emptive changes) accounts for maintenance that goes beyond the goals of the original development effort.

Maintenance is the most time consuming and costly phase of the software development lifecycle. For every dollar spent on creating a new software system, nine dollars is spent on maintaining it throughout its useful life. By the late 1980s maintenance spending accounted for an estimated $30 billion worldwide. At that time programmers working within an organization typically spent over 55% of their time performing maintenance [Cor 89]. By 2020 it is expected that over 67% of all programmers will be working exclusively on maintaining pre-existing software systems [DKV 99]. Any activity that even minimally reduces maintenance efforts would yield significant cost savings within the software industry [CC 90].

One reason why maintenance is such a prevalent activity is the sheer quantity of software that exists. In 1999 the total volume of software was estimated at 640 billion statements

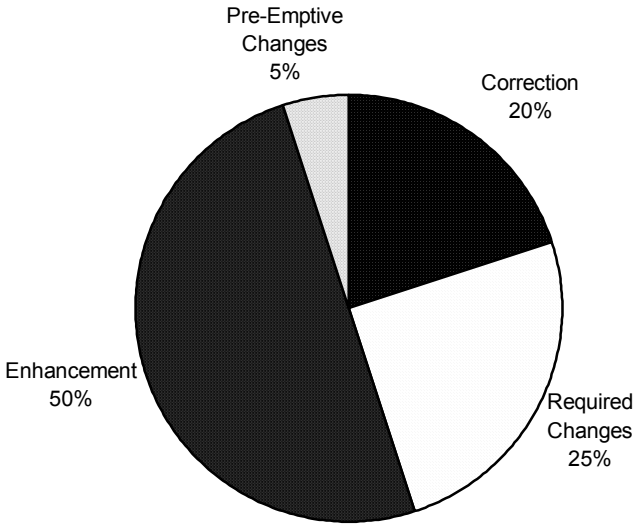


Figure 1: Types of Maintenance

[DKV 99]. Based on this figure and the software quality expectations of Jones [Jon 98], an estimated 35 billion programming errors stemming from all phases of the development lifecycle remain in code that exists today [DKV 99].

The abundance of legacy information systems is another reason why maintenance is such a widespread practice. A legacy information system is defined as "any information system that significantly resists modification and evolution" [BS 95]. Bisbal, Lawless, Wu and Grimson [BLW+ 99] have identified several problems with legacy systems:

- The costs for maintaining them are very high.
- The hardware a legacy system runs on is often obsolete.
- Compared to modern standards, legacy systems are often very slow.
- There is often a lack of documentation and very little detailed knowledge on how the system operates.
- Legacy systems are difficult to interface to other systems.

- Legacy systems are not usually extensible.

Many businesses such as banks or insurance companies have legacy systems that perform mission critical tasks. The systems are so important that their failure would have a very serious impact on the operations of the business. It is vital to keep these systems running. Yet ever-changing business practices and technical requirements such as Year 2000 compliance necessitate modification.

An example of the need for maintenance is exemplified in the use of the legacy programming language COBOL. The 40-year-old language [Bra 99] is commonly regarded as outdated. Nevertheless, COBOL is still widely used in industry. According to a widely cited report by Gartner Inc. [Bro 01, Ulr 01, Wil 01], COBOL is used for storing 85% of the world's data. Around 200 billion lines of COBOL code existed in the year 2000. Over the next four years it is expected that an additional 20 billion lines of *new* COBOL code will be added to this figure. In North America alone, 90,000 programmers work in COBOL maintenance and development. The use of Cobol to facilitate *Customer Information Control Systems (CICS)* transactions on the Internet is widespread. According to IBM, more Cobol/CICS transactions are executed in a day than the total number of web page hits [Wei 01].

2.1 Maintenance Activities

Given the critical importance of many software systems and the inevitability that maintenance will involve making changes, how can this be accomplished? Five options for maintainers are summarized in Table 1. We distinguish maintenance activities by how they affect internal and external properties of an existing software system. Internally we refer to the source code as the fundamental system component. For the sake of this discussion, the observable behaviour of the system and the functionality it provides (i.e. the 'feature set' of the system) are the external

		Software System Properties		
		Source Code	*Behaviour*	*Functionality*
Maintenance Activity	*Redevelopment*	Discarded	Newly Created	Newly Created
	Wrapping	Unchanged	New Interface	Unchanged
	Migration	Modified	Modified	Unchanged
	Restructuring	Modified	Unchanged	Unchanged
	Reengineering	Modified	Modified	Modified

Table 1: The Impact of Maintenance Activities on Software System Properties

properties we refer to. Each of the maintenance activities listed in the left column impacts one or more of the system properties as indicated in the columns on the right. A discussion of each of the five maintenance activities follows.

2.1.1 Redevelopment

Redevelopment involves replacing an existing system with a new one [BLW+ 99]. One advantage of this method is that it provides an opportunity for a new system to be designed and developed from the ground up. Many of the problems associated with legacy systems are ameliorated with new equipment and freshly minted software. The problem is that newly developed software often has a lot of bugs in it. Even after a thorough cycle of testing, some bugs do not become apparent until well after the software is put into actual use. At some point in time the old system is retired and the new system replaces it. Often there is no way to revert back to the old system. In a mission critical environment, any failure of the system can be catastrophic, leading for instance to loss of life or a severe disruption of business operations. The high risk of failure is often unacceptable for redevelopment to be considered a feasible means for changing a system.

2.1.2 Wrapping

Wrapping is commonly implemented as a means for dealing with legacy systems [BLW+ 99]. It involves the encapsulation of an existing system with an outer shell, which acts as an interface to the system it encloses. Users see the system only from the front end of the shell, which usually consists of a graphic user interface that permits specific operations. Wrapping is usually a short-term solution that provides better access, but does nothing to address the other problems with legacy systems mentioned above. One of the drawbacks of wrapping is that it actually increases maintenance costs. This is because the original system and the wrapper software must be maintained.

2.1.3 Migration

Migration endeavors to keep the data and functionality of the original system while shifting it to a more adaptable and maintainable environment [BLW+ 99]. This method is advantageous for a number of reasons. It is a long-term solution for dealing with legacy systems. From a functional and economic perspective, the outcome of the migration yields a system that has the potential for resolving all the problems with legacy systems mentioned above. Unlike wrapping, migration does not mask an existing problem; it evolves the system into more maintainable, problem-free state. Unlike redevelopment, migration incrementally leads to a new system, thereby reducing the risk of catastrophic failure. A well-planned and implemented migration should not cause a severe disruption of mission critical systems. Nevertheless, the main drawbacks of migration are that the process is complex and it takes longer to achieve. Migration requires a disciplined approach to program and database understanding before changes are introduced into the system. A variety of strategies for populating databases, testing the new system and handling cut-over are required.

2.1.4 Restructuring

Maintenance can involve making internal changes without affecting the external behaviour of the system. Chikofsky and Cross [CC 90] define *restructuring* as:

> "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)"

Cleaning up code to make it easier to read, implementing structural improvements or making changes to conform to a particular coding style are all examples of restructuring activities. In this way, restructuring can be characterized as preventative maintenance for the software system.

2.1.5 Reengineering

Reengineering, also known as *renovation* or *reclamation* is defined by Chikofsky and Cross [CC 90] as:

> "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

Unlike migration, which focuses on keeping the data and functionality of the original system, reengineering involves making broad changes that transform the existing system into a new one. In general, reengineering is considered a very difficult and highly complex task. Bergey *et al.* [BST+ 99] suggest that software reengineering is at least as difficult as designing and implementing a new system from scratch. Failure is a common result:

> "Reengineering efforts are replete with examples of failures. In fact, the documented record suggests that there are far more failures than there are successes … There are just as many failures in trying to evolve systems as there are building them in the first place."

A process for carrying out software reengineering tasks outlined by Kazman, Woods and Carrière [KWC 98] is shown in Figure 2. It consists of a horseshoe shaped model for the steps undertaken to transform a system. Starting with original source code on the lower left side, a series of steps extract details and abstract information. This is known as *architecture recovery*

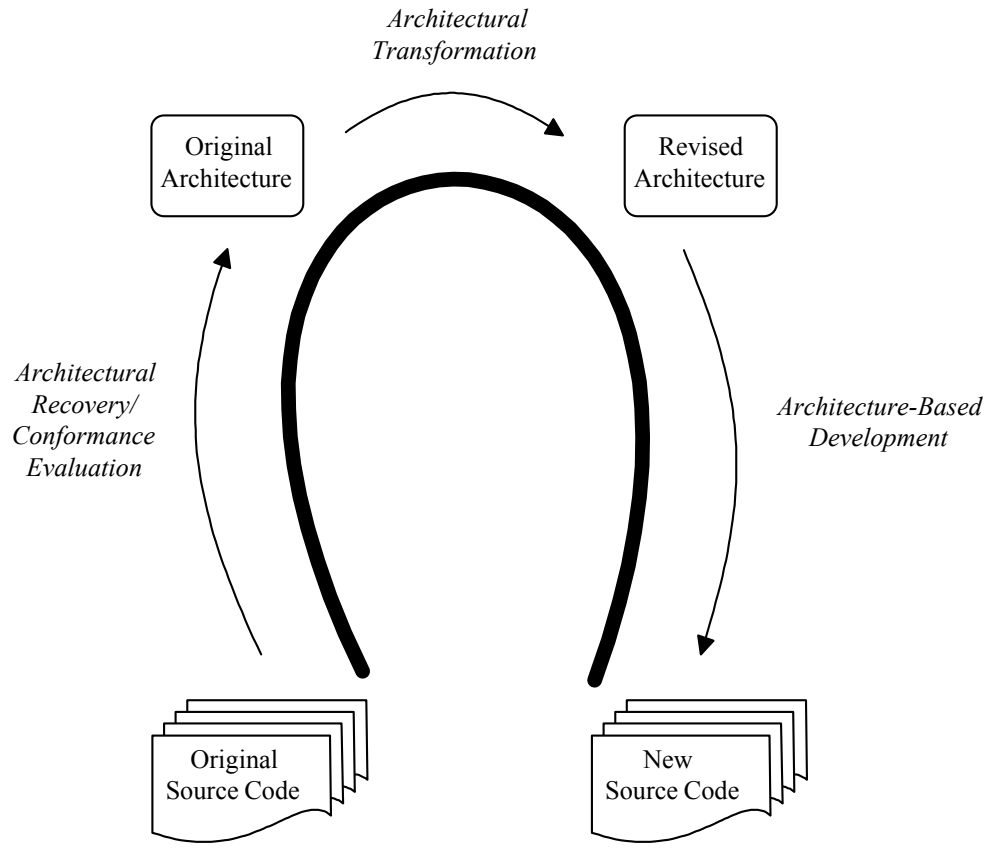


Figure 2: A Horesehoe Model for Software Reengineering Tasks

---

and conformance evaluation. The goal is to gain a high level understanding of the existing code and its conformance to the original design intent. Once this level of program understanding is accomplished, adjustments to the architecture can be outlined. This is known as *architectural transformation*. This "bridge step" brings the process from the left side over to the right side of the horseshoe. Based on the revised architecture, development efforts work down the right side of the horseshoe towards the goal of creating source code for the new system. This is known as *architecture-based development*.

3 Reverse Engineering

In Chapter 2 we typified software maintenance, demonstrating its necessity and outlined five paradigms for changing systems. The ultimate goal of maintenance is to facilitate change in a system, however large or small, that does not bring about adverse side effects. This is a challenging task that requires the maintainer to have a good grasp of the system being altered. Reverse engineering is the key endeavor within the maintenance activity that provides the knowledge maintainers need to effectively do their jobs.

The term reverse engineering has it roots in the analysis of hardware. In this context, Rekoff [Rek 85] defines it as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system." In a software context, reverse engineering is about system comprehension at a functional and behavioural level [TPS 96]. Functional comprehension provides insight into *what* the system does. For instance, a software maintainer might be interested in finding out if a payroll system automatically deducts pension contributions along with required deductions for taxes and insurance premiums from employee pay cheques. Behavioural comprehension highlights *how* a system works. Continuing with our example, the maintainer might be interested in determining if pension plan deductions are calculated using a formula, or obtained from a table using the employee's income as a key.

It is important to note in these examples that the maintainer may not necessarily have knowledge of the domain in which a system operates. A programmer who is required to make changes to a payroll system may not have a background in accounting. At the same time, it is likely that accounting managers do not have experience with the dynamics of software and the challenges involved in maintaining the software for a payroll system. For this reason it is

important to consider the comprehension of a system's application domain as an integral part of the work role for software maintainers.

A study by von Mayrhauser and Vans [MV 92] indicates that in industrial practice, maintainers use a variety of strategies for comprehending systems. Lethbridge and Anquetil [LA 97] examined the daily work routines of programmers and system analysts, revealing that much of their time is spent gaining an understanding of how the system works as a precursor to making changes to it. Most often the concern is only focused on understanding the portion of the system that is relevant to the change being introduced. After the change is made, concern focuses on changing another portion of the system. As a result, an overall understanding of the function of the system is never developed on the part of those who make changes to it. For this reason, tools that support *Just In Time Comprehension* of source code for very large software systems would be highly beneficial.

3.1 Objectives

Chikofsky and Cross [CC 90] have identified a number of objectives for reverse engineering. Very large systems in particular are difficult to maintain because of their sheer size and the complexity of their structure. Reverse engineering endeavors to provide the tools and processes to help maintainers cope with the complexities of large systems. One way of dealing with complexity is to find alternate ways for representing information about a system to maintainers. The goal is to present a simplified view of system in terms of specific characteristics. In particular, graphical views have been effective as and aid to comprehension.

The development process is essentially a collection of decisions that are made on how a solution that satisfies a set of requirements is formulated. It is often the case that after a project is completed, these decisions and the original intentions of the developers are lost. This is

especially true with legacy systems where it is often the case that a great deal of time has passed since the system was originally implemented. Source code is an artifact whose analysis in a reverse engineering context provides a means for recovering lost information about the system and the reasoning behind it's original construction.

As we mentioned in the previous section, maintenance is really about managing change in a software system. One objective of reverse engineering is to assist in the identification of negative side effects that changes to a system might bring about. Effective mechanisms for fostering comprehension and evaluating the effects of change can go a long way towards ensuring the integrity of the system being maintained.

One positive side effect of system comprehension is the identification of generalized solutions for specific problems. For instance, continuing with our payroll system example, consider the situation where deductions for taxes and insurance premiums are calculated through the use of a lookup table. A maintainer interested in adding the ability for the system to handle pension plan deductions can make use of a lookup table similar to that used for taxes and insurance premiums. In this way, reverse engineering facilitates the reuse of a previously thought-out solution for a specific requirement of the payroll system.

One of the most important objectives of reverse engineering is to provide a means for extracting and representing information about software at various levels of abstraction. At the lowest level source code is usually represented as a tree or graph structure that corresponds closely to the internal representation found in most software compilers [HWS 00, KWC 98]. Concern is focused on extracting information such as control flow, global variables, data structures and resource interactions.

At a medium level of abstraction, behavioural, procedural or modular features of the system might be the concern. For example, behavioural characteristics such as the usage of memory, uninitialized variables, ranges of values and plans for algorithms might be extracted [MWT 94].

At a high level of abstractive detail architectural characteristics are typically represented. Business rules, policies, responsibilities [MWT 94] and requirements [DKV 99] may also be characterized. Woods, Carrière and Kazman [WCK 99] and Riva [Riv 00a] contend that architecture can be thought of as a collection of design decisions that provide an overall vision of the functional requirements of a system. The implementation process translates the architectural vision into functioning code. If an implementation correctly follows the architectural vision, then mappings will exist from the architectural concepts to the code produced. These mappings represent reasoning on the part of developers on how to translate the architecture into a working system. This information is usually not recorded in the source code nor is it often available directly from the original developers or from available documentation [BKV 96b, MJS+ 00]. Architectural recovery is typically a manual process [HWS 00] that involves analyzing code and making an attempt at coming up with reasons why the system was constructed the way it was. For instance, a process known as *lifting* involves the induction of file information from functional details about a system [WCK 99]. The recovery of architecture is comparable to archeology [GAK 99]. In both fields the goal of researchers is to recover meaning from artifacts that they are presented with.

Knowledge of a system from an architectural perspective provides the maintainer with much more useful information than that which would be obvious through source code inspection. This is especially true in the case of very large systems where a complete examination of the entire source code base is not possible.

3.2 Software Representation

Reverse engineering activities involve the collection, manipulation and analysis of software representations. Graphs are typically used for these purposes for a number of reasons. At an intuitive level, graphs effectively present intangible concepts. Software is not a physical entity, so graphs provide a means for analyzing and manipulating software from a visual perspective. Other properties make graphs an effective instrument for representing software.

Graphs can be *constrained*. Rules based on the application domain can be applied which dictate how a graph is constructed. For instance, in a relational graph two vertices may not be connected by two edges of the same type [EKW 99]. In certain circumstances, these constraints provide a formal means for verifying the validity of a graph.

Graphs are often used as a way of modeling relationships [PT 90]. In this context, nodes are representative of concepts or objects that are relevant to a particular problem domain. Edges are used to establish relationships among concepts or objects that have been defined. Other properties such as names or line numbers that describe entities and relations can also be added to the graph. These are commonly referred to as node or edge attributes. This *Entity-Relationship (E-R)* model [Che 76] provides a powerful representational framework that can be applied to scores of application areas within the reverse engineering domain. For instance, a call graph is commonly used by maintainers to identify source code dependencies. In such a graph nodes are representative of program modules. Edges between the nodes represent module invocations. The resulting graph provides a visual indicator of the interconnectedness of the software being evaluated. By simply changing what the nodes and edges represent (i.e. changing the node and edge semantics), new models can be created. In a state transition graph the nodes and edges are used to respectively represent states and transitions among states. A Program Evaluation and

Review Technique (PERT) chart represents nodes as activities with edges representing orderings for the activities.

E-R models are important in the context of reverse engineering for two reasons. First, they provide a clean separation between the information that defines the allowable characteristics of a graph and the data that is represented by a graph. The former is known as *schema* and the latter as *instance*. A graph schema defines the constraints and node and edge semantics for a class of graphs [BGH 99]. An instance is data represented as a graph constructed in accordance with the graph schema.

Second, E-R models are one of the common techniques used to outline the structural characteristics of databases [Som 01]. This quality is of particular importance to the reverse engineering community. Reverse engineering analyses typically involve the storage and manipulation of extremely large quantities of information. Using E-R models to structure the graphical representation of software aids significantly in defining database structures that support reverse engineering analysis efforts. So any graphical representation of software is really based on an E-R schema that can be represented as a database. Within the reverse engineering community these are referred to as repositories or design databases [Cor 01].

4 Tool Support

Chapter 3 provided an introduction to the goals of reverse engineering practice within the software domain. In this chapter we focus on characterizing tool support for reverse engineering. We discuss how tools are constructed, how they are used and provide insight into research issues that have had an impact on the adoption of reverse engineering tools in industry.

4.1 Reverse Engineering Activities Supported

Software engineers carry out a number of different activities to achieve the maintenance goals outlined in Chapter 2. Only small subsets of these activities are supported by reverse engineering tools [Til 98]. In many instances, a tool provides assistance but still requires significant manual intervention on the part of the user. Research in automated techniques is ongoing but has largely been unachieved.

Much of the achievement in tool support for reverse engineering has been in the area of code analysis. At this low level of abstraction the source code is generally examined from a syntactic perspective to support the following activities:

- **Disintegration**. Breaking larger systems into subsystem components.
- **Pattern Matching**. Identification of instances within the source code where an identical coding pattern occurs.
- **Program Slicing**. Isolation of all code that relates to or in some way impacts the execution of a specific point in the source code.
- **Dependency Analysis**. Evaluation of the reliance of system components on other internal or external components.

- **Metrics Evaluation**. Measurement of the code according to accepted standards for various characteristics such as size, complexity, quality, maintainability, etc.
- **Exploration**. Support for navigation throughout the source code.
- **System Visualization**. Generation of views for examining the system visually.

Müller et. al. [MJS+ 00] argue that code analysis provides incomplete information about a system because the source code artifact does not contain everything there is to know. For instance, architectural concepts, design decisions, tradeoffs and knowledge of the application domain are all missing from the source code. It is difficult to make changes to a large system without this kind of high-level abstractive detail.

One solution to this problem involves the adoption of a continuous program understanding process [MJS+ 00]. In such a practice, emphasis is placed on recording knowledge about changes made to a system and establishing relationships between this information and the code affected by it. The benefit of this process is the ability to trace the historical evolution of the code from the original developers to its current state. Over time the value of this chronological record would increase significantly. Though managers, developers, maintainers and others with knowledge of the system might change, a persistent chronicle of accumulated information about the system would exist for anyone involved with it in the future.

4.2 Tool Construction

Although many reverse engineering tools exist, most feature the same underlying architecture [CC 90] and operate in a similar fashion. In general, reverse engineering tools consist of the following three components [BGH 99, Dev 99, God 01, HWS 00, WOL+ 98]:

1. **Information Extractor**. The front ends of reverse engineering tools typically input the source code and extract information from it. A lexer reads the code and breaks it into

lexical tokens. These tokens represent the keywords and basic building blocks for the program based on the specific programming language that is being used. Next, a parser groups the lexical tokens into programming constructs like statements, expressions, declarations, etc. A tree structure called an *Abstract Syntax Tree* (*AST*) is used to represent these constructs. Some tools make use of a semantic analyzer to add additional information about symbols such as scopes, types and values to the AST. These comments change the structure of the AST to yield an *Abstract Semantic Graph* (*ASG*).

An example from [LL 00] demonstrates the differences between an AST and an ASG. A small C++ program is shown in Figure 3. The corresponding AST representation for the program is shown in Figure 4. The tree consists of a hierarchy of nodes representing source code entities. Entities at the top of the hierarchy are decomposed into various sub-components via edges that identify the relationship between each of the nodes. The ASG is shown in Figure 5. Dashed lines indicate where changes from the AST have been made. Two differences between the AST and ASG are observable in this example:

- In the AST, multiple nodes with the same type exist. In the ASG, only one node for each type is permitted, so multiple instances of the same type nodes are combined into a single type node. In the Figure 3, two different instances of `object(x)` have been created (one in class `A` and one in function `main`). Both are instances of type `int`, so both have an `instance` edge leading to the same `type(int)` node.
- In the AST, references by name to objects and by types to classes are permitted. In the ASG, these indirect references are removed and replaced by direct references to the corresponding objects and classes. In the Figure 5, we see `object(y)` now shown as an instance of `class(A)`. The `opd(1)` and `opd(2)` edges that referred to

```
Class A
{
  public:
    int x;
};
main()
{
  int x;
  A y;
  x = 0;
  y.x =1;
}
```

Figure 3: A Small C++ Program

`name(x)` and `name(y)` nodes in Figure 4 now refer directly to `object(x)` and `object(y)` nodes respectively.

2. **Repository**. The quantity of information extracted from source code can be substantial. For this reason, information extracted from the source code is typically organized and stored in a database rather than preserved in memory.
3. **Analyzer/Visualizer**. The information in the repository is processed and analyzed with the results presented visually or through reports.

A number of reverse engineering tools exist in the research community. Taking a look at only information extractors, many differences become apparent [God 01]:

- Some are better than others at extracting facts from source code.
- The languages evaluated by each information extractor are limited and different from each other.
- The level of abstraction varies considerably. For example, some extract facts at the code level, while others return code modules, functions and procedures.

Repository managers also vary in their ability to assist in handling large quantities of data.

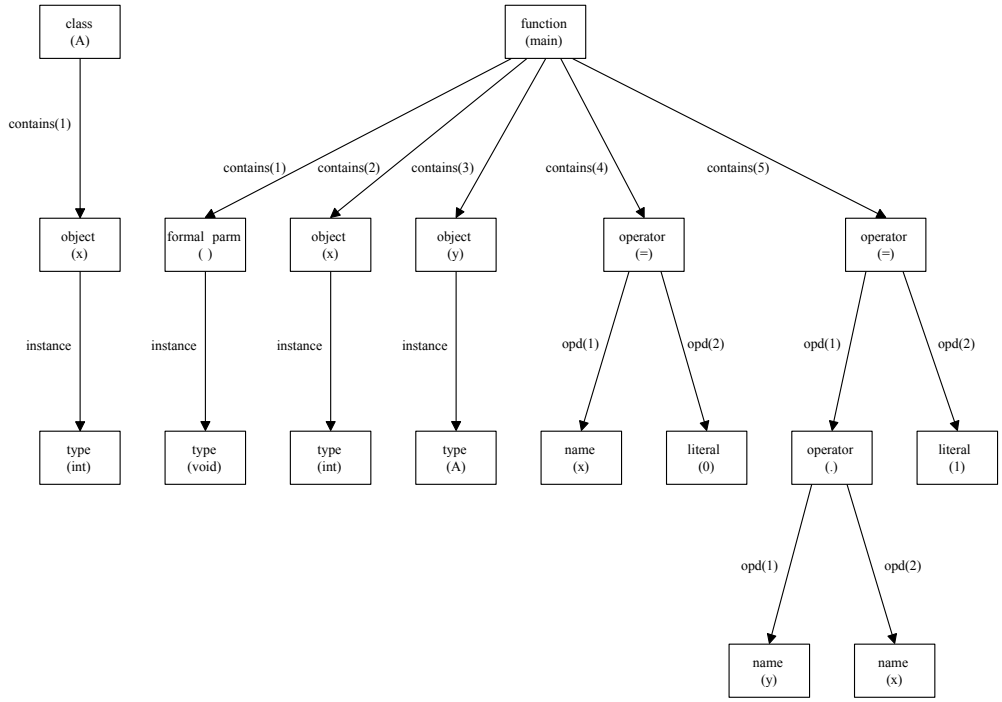


Figure 4: The AST for the Program in Figure 3

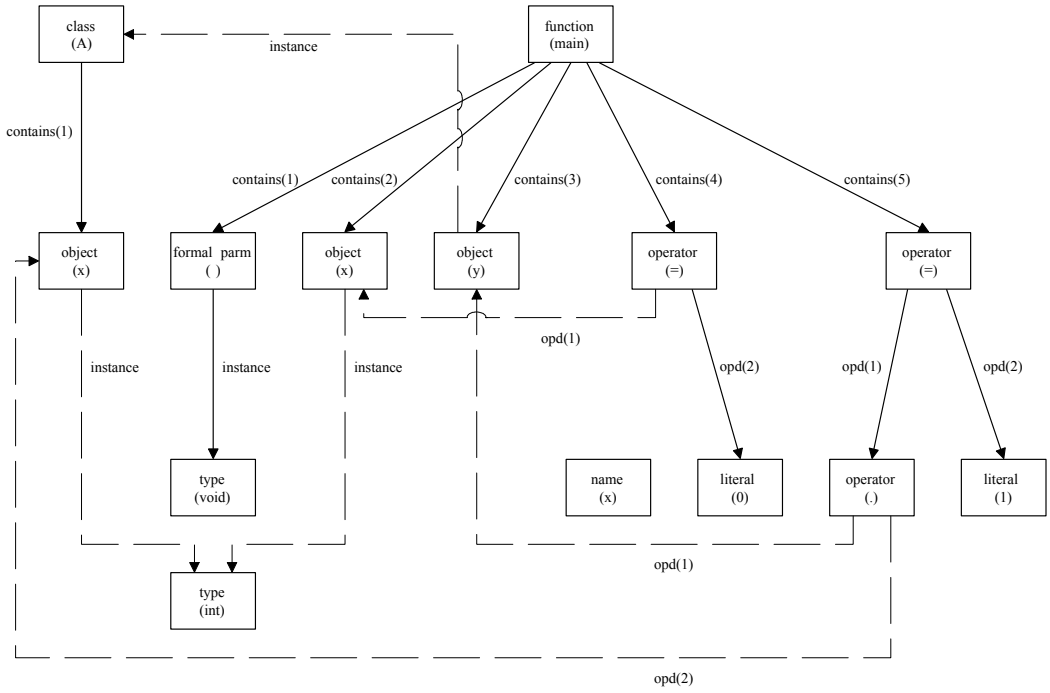


Figure 5: The ASG for the Program in Figure 3

4.3 Tool Usage

Despite the existence of a number of reverse engineering tools, not much is known about how maintainers use them. Few empirical studies exist that document what is known as the cognitive aspect [TPS 96] of reverse engineering. This refers to the approach taken by maintainers towards understanding and managing the complexities of large software systems. Research has explored how different tool designs and capabilities can aid in program comprehension [SWF+ 96, SFM 97, SWM 97].

Lethbridge and Singer [LS 97] have researched the practices and needs of software engineers who perform maintenance tasks on large-scale telecommunications software. In particular, they were interested in determining which tools were being used and for what purpose in relation to maintenance tasks. Results were obtained using questionnaires, interviews, observations and logs that recorded the use of tools automatically. Although the study could hardly be considered representative of the work processes of software maintainers in general (only about a dozen software engineers who work within the same group were involved), the results provide a unique insight into what maintainers do and what they need to help them do their jobs better.

There is a significant need for tools that assist maintainers in searching through source code. A number of tools were already in use for this purpose. Positive comments were that the tools were easy to use, provided functionality that was useful and provided the results quickly. Lack of tool integration and incompatibility was the leading complaint about reverse engineering tools voiced by software maintainers. Other negative comments centered on the fact that needed features are nonexistent and those that do exist are not powerful enough. Features are important as Müller et. al. [MJS+ 00] estimate that less than 20% of the capabilities of a reverse engineering tool are used 80% of the time.

4.4 The Need For Integration And Interoperability

While progress has been made towards increasing the performance and usefulness of reverse engineering tools, most continue to exist in isolation, lacking any means for sharing information among each other [EKW 99, Per 00]. Lack of integration and interoperability has become a major barrier to the advancement of a cohesive reverse engineering environment [LA 97, WOL+ 98]. For example, early adoption of the C++ programming language was hindered by a lack of integrated tools to assist in the development and maintenance of C++ code [RW 91].

A myriad of reverse engineering tools exist, each with a specific strength or specialized application area [Let 98]. Analysis from different tools can help speed up the reverse engineering process [LA 97, Riv 00a]. Holt *et al.* [HWS+ 00] lists eighteen reverse engineering, software modeling, analytical and graphing tools that provide many different types of analyses that reverse engineering practitioners make use of. Maintainers would like to leverage their results by combining the output of different analyses from different tools [BGH 99, God 01, KWC 98, MWT 94, Nag 96]. Without an integrative component, maintainers are forced to work independently with each tool starting from scratch [BGH 99, KWC 98]. Manually integrating results from different tools is tedious and time consuming [DRW 96, Per 00, Riv 00a]. Bergey *et al.* [BST+ 99] describe how integration problems seriously hindered a major software reengineering effort:

> "The new strategy involved reengineering current code to a scaled down set of requirements … The environment consisted of a set of proven tools, each of which was well regarded as among the leaders in its class. However, the plan depended on integrating the tools in a seamless way and on using this environment for production of the system … The integration did not work … the project lost precious time and millions of dollars in getting back on track. The lesson is that … even when individual components are proven, integration aspects can come back to haunt a project"

Considering these results, it is not surprising that reverse engineering tools have a low level of adoption among maintainers [MJS+ 00].

5 Integration Approaches

So far we have demonstrated the need for integration and interoperability among reverse engineering tools. But how can this be achieved? A variety of integration standards exist including Microsoft's *Object Linking and Embedding (OLE)* [Cha 96], Java's serialization interface [GJS 96] and the Object Management Group's *Interface Definition Language (IDL)* [OMG 97]. Although these standards facilitate the exchange of structured and unstructured information among programs, they have not been applied in the reverse engineering community for one or more of the following reasons [BJK+ 00]:

- They operate only on particular hardware or software platforms.
- They are unnecessarily complex.
- They do not represent and transfer data in an efficient manner.
- They are not language independent.
- They lack extensibility in providing a means for storing extraneous information in addition to the standard information being exchanged.

Without the benefit of an existing standard to work with, members of the reverse engineering community have embarked on creating their own solutions for tool integration. Three approaches have been pursued: Intermediate Representations (IRs), Application Programming Interfaces (APIs) and Standard Exchange Formats (SEFs).

5.1 Portable Intermediate Representations

As was mentioned in Section 4.2, the front-end activities of reverse engineering tools are very similar to compilers. Both process source code yielding a tree (an AST) or graph structure (an ASG) that organizes lexical, syntactic and semantic information. What happens next is what

differentiates them from each other. A compiler carries out a number of tasks with code generation as the goal. A reverse engineering tool carries out a number of tasks with analysis and generation of high-level abstraction as the goal. Needless to say, the construction of an AST or ASG is an integral part of the process. The tree or graph structure is known as the *Internal Representation (IR)* that corresponds to the source code input.

One of the early ideas for enabling integration among reverse engineering tools involved storing the IR so that it can be shared among different tools. The idea behind exchanging IRs is not new. Programs for writing compiler IRs to a file have existed for some time. In the compiler domain they are known as *picklers* [BJW 87], creating what is referred to as a *Portable IR*.

Within the reverse engineering domain, portable IRs never really caught on because of a number of fundamental problems. First, there was no standard IR format that compiler picklers wrote to. This made it difficult to select an IR appropriate for the reverse engineering community. According to Rugaber and Wills [RW 96], there were too many IRs to consider. Some examples are *ANDF* [OSF 91], *ASFIX* [BKO 99], *CCG* [KM 94], *DIANA* [GW 81], *Eli* [GHL+ 92], *Hoof* [ADH+ 99], versions of the *Interface Description Language (IDL)* from Lamb [Lam 87] and the Object Modeling Group [OMG 91], *IRIS* [BFS 88], Jordan's *Modula-3* IR [Jor 90], *Ponder* [GA 95] and the *Slim Binary Format* [FK 96].

A second problem was that IRs are not language independent. They are built to represent only a specific programming language. IRs lack the generality and extensibility that a reverse engineering representation requires [Kie 01]. Changing the syntax of an IR so that it will support the representation of another programming language is a very difficult task.

Third, IRs fall short in terms of interaction, representation and performance [KGW 98]. Reverse engineering tools often feature interactive environments where users can generate

different views of a system based on different analyses. Because of their fixed nature, these views are impossible to create with IRs as the representational foundation. Support for the representation of software at multiple levels of abstraction is a key requirement for reverse engineering environments. IRs only represent source code at a low level of abstraction which severely limits their applicability in reverse engineering. A high level of performance is a necessity when handling the analysis of millions of lines of source code. Unfortunately, recent advances in optimizing compiler technology have not been passed on to IRs. This makes it difficult to make use of IRs for analyzing large-scale software systems.

5.2 Application Programming Interfaces

An Application Programming Interface (API) refers to a method whereby one program provides the means for another program to interface to it. According to Sim [Sim 00a] there are three approaches by which APIs can be used to integrate reverse engineering tools:

Library. One tool provides functions that other tools use to access it.

Communication Protocol. CORBA or some other peer-to-peer or client-server architecture is used to manage exchange among tools.

Hybrid. A combination of both the library and communication protocols is employed. Tool developers agree on a library of functions that each will make available and on the protocol each will use to exchange information between them.

The primary advantage of using APIs to integrate reverse engineering tools is that dynamic inter-tool communication increases performance. Tools communicate directly with each other, eliminating the overhead involved in writing to and reading from external files. Disk storage requirements may also be reduced. In a non-integrated environment, each tool independently

stores data in their own proprietary format. Well-integrated tools eliminate redundant repositories, as one repository is used freely among all the tools in use.

There are some disadvantages to using APIs for reverse engineering tool integration. Different tools have different features that users may want to take advantage of. Although the use of APIs significantly improves the speed and ease of interaction among tools, they still need to know *how* they can interact with each other. A tool must be aware of the requests it can make of another tool it interfaces with. Likewise, each tool must be aware of the structure of the data that is exchanged between them [LA 97]. Coming to an agreement on how two tools will interact is challenging. The situation is further complicated by the fact that API functionality is typically built into the code for each tool.

5.3 Exchange Formats

An exchange format arises from an agreement made between tool developers on the syntax and semantics of information to be exchanged between two or more reverse engineering tools. Syntax refers to the structure of the information contained in the exchange format. Semantics relates to the meaning of the information being exchanged. A schema is typically used to define the information that the exchange format can consist of and how it should be interpreted [KCE 00b]. Files are typically used to facilitate the exchange, although Internet-based document exchange technologies have been used for this purpose as well.

There are a number of advantages to using exchange formats to integrate reverse engineering tools. An exchange format is typically much simpler to implement than a portable IR or an API. Often a simple converter is all that is necessary to get information stored from one tool into the format supported by another tool. Typically the information exchanged is represented in a textual format so it is human readable. Since an exchange format is based on an agreement between two

or more parties, there is very little ambiguity in the software representation. Everyone is aware of what each entity stands for in the format. Unlike portable IRs, an exchange format is not tied to representing software written in a particular programming language.

The main disadvantage of exchange formats is similar to that for APIs. It is difficult to get all those interested in using the format to agree on what information it should transfer and how it should be structured. The process is time consuming and changes are difficult to manage among all interested parties. The extensibility of an exchange format only partially mitigates this problem. Although the addition of proprietary extensions to an exchange format is easily accommodated, managing this extraneous information can be a challenge. Other tools that do not make use of the extensions still need to maintain them as part of the information exchanged. The integrity of the extensions can be affected by changes that another tool makes to information in the format.

Another disadvantage of exchange formats is the overhead involved in writing to and reading from files. Reverse engineering analysis often involves the manipulation of information relating to millions of lines of source code. At this level of magnitude, file management and in-memory storage issues [KCE 00b] significantly affect the integrative performance of reverse engineering tools.

5.4 The Need for a Standard Exchange Format

The lack of a consistent model for the structural makeup of software representations is a major barrier to reverse engineering tool integration. Such a model would eliminate data redundancy and provide a representational foundation for tool developers interested in integrating their tools with others [KCE 00b, LA 97, Let 98]. Within the reverse engineering community this model is referred to as a *Standard Exchange Format (SEF)*. Unlike the practical approaches that IRs, APIs

and various exchange formats provide, the development of an SEF is a more abstract goal. It lays out a structural map for software exchange information that would be applicable to all integration approaches [Sim 00a].

The establishment of an SEF would also facilitate the creation of source code repositories for program understanding researchers. This collection of 'software guinea pigs' [Hol 00, MRE 01] would contain standardized SEF-structured representations of software extracted from various systems. Such a collection would eliminate the labor involved in parsing and pre-processing source code for research purposes [Let 98].

6 Exchange of Software Representations

In the previous chapters we have established reverse engineering as an integral activity for software maintainers. In general, tools provide assistance in carrying out reverse engineering tasks in isolation. The creation of an integrated reverse engineering environment has been hindered by the lack of an element that enables the exchange of software representations among different tools. It has been argued that a standard format should be created to make interoperability among reverse engineering tools possible.

In this chapter we take a look at fourteen formats that have been used in the reverse engineering community to exchange software representations. Most of the formats have a very specific application area, for example, enabling exchange among programs that make up a particular toolset. Nevertheless, it is useful to review their characteristics to understand how each format achieves its exchange purpose.

6.1 Characteristic Properties

To aid in our review of exchange formats we summarize their characteristics using various categories for classifying their properties. A description of each of these categories follows.

6.1.1 Abstract Syntax

One of the most important properties of an exchange format is the data structure used to represent software. This is referred to as the *abstract syntax* for the format. In relation to software exchange formats, the abstract syntax can be divided into three general categories: *structured data*, *trees* and *graphs*. Each of these is outlined as follows.

Structured Data. The format uses a structure for data that is not a tree or a graph. The arrangement of the data exchanged may be proprietary or very simple, consisting only of atomic units.

Trees. The format exchanges tree structures but *not* graphs. Although trees are graph structures, this category distinguishes formats that cannot represent graphs in general from those that can.

Graphs. The format exchanges various graph structures including trees. These formats may be capable of representing many different types of graphs, but not necessarily all types of graphs. For the purpose of this review formats that support the following graph characteristics are identified:

- *Typed*. Support for a classification scheme for nodes and edges is provided.
- *Attributed*. Attribute labels consisting of strings or values can be associated with nodes or edges.
- *Inheritance*. Nodes and edges can take on the properties of other nodes and edges in the graph.
- *Hierarchical*. Support for the representation of graphs within graphs is provided.

6.1.2 Levels of Abstraction

As we mentioned in Chapter 3, one of the primary goals of reverse engineering is to represent software at various levels of abstractive detail. Consequently, it is important to consider the level of abstraction that an exchange format is designed to convey between tools. Three levels of abstraction are considered in this review:

Low. Program statements from the source code are represented as structured data, most often organized into an AST or ASG representation.

Medium. Program components above the statement level are represented. Some examples are functions, types and global variables.

High. Architectural entities such as modules, subsystems, classes and packages are represented. Other characteristics related to the architectural development of a system such as plans, policies, requirements, business rules and responsibilities may also be recorded.

It should be noted that within the reverse engineering community there is some contention as to what is represented at each level of abstraction. For instance, Kazman, Woods and Carrière use a four-layer nomenclature that distinguishes between source code and AST/ASG representations [KWC 98, WCK 99]. Our categorization scheme is based on a view of abstraction that is commonly accepted among reverse engineering practitioners [God 00a, God 00b, HWS 00, MWT 94].

6.1.3 Type of Encoding

In general, two encoding types are used by exchange formats:

Text. All data and any formatting statements such as tags or labels are exchanged in plain ASCII text. One of the advantages of textual exchange is that it is human readable and can easily be edited in a standard text editor. The main drawback of this type of encoding is the size of the data being transmitted. No effort is made to compress the information being exchanged.

Binary. A compression algorithm is used to convert textual information into a binary format for exchange. The main advantage of this method is that the size of the transmission is substantially reduced. A binary encoding is much more compact than text. Nevertheless, the overhead involved in encoding and decoding information at each end of the transmission can affect the performance of the exchange between each tool. It also requires that each tool use the same method for encoding and decoding data.

6.1.4 Transfer Mechanism

The transfer mechanism is a characterization of the method used carry out data exchange. The following categories are distinguished:

File. One tool writes data to a file. Another tool reads data from the file. Simplicity is the obvious advantage of this transfer mechanism. The main detractor is scalability. Large files are difficult to manage and the overhead involved in accessing them is often unacceptable.

Structured Text Stream (STS). Text stored in a structured format is packaged and exchanged from one tool to another through a communications medium such as the Internet. Within the reverse engineering community *Hypertext Markup Language (HTML)* [WWW 99] was originally thought of as a means for facilitating exchange [TS 97]. Since then, more flexible technologies such as *CASE Data Interchange Format (CDIF)* [Ern 97, Lem 98, PKS 98] now known as *XML Metadata Interchange (XMI)* [IBM 00, OMG 98, DBI 00] and *Extensible Markup Language (XML)* [Cov 98, WWW 00, ZK 01] have been used. XML in particular is an emerging standard that has been widely embraced by many academic and commercial software developers. The notation is verbose with lots of HTML-like tags but is readable by humans and automated processors. A convenient distinction is made between the structure of data to be transmitted and the data itself. Structural characteristics of the data are defined in an XML *Document Type Definition (DTD)*. An initial transmission of the DTD outlines how instance data in subsequent transmissions will be structured. This partitioning between structure and data provides a flexible environment for exchanging information that is gaining popularity among tool integrators.

Direct Inter-Tool Functionality (DIF). Exchange between tools is facilitated through direct tool-to-tool interfaces. The type of interface may be local (for example, the communication

between two executing processes) or wide ranging (for example, client-server communications across an Intranet or the Internet). DIF integration is typically "hard-coded" into each of the tools involved. It generally offers high performance at the cost of flexibility.

6.1.5 Schema Type

As we mentioned in Section 5.3, a schema defines the information that an exchange format can consist of and how it should be interpreted [KCE 00b]. In our review of exchange formats we distinguish the schemas for each of the exchange formats as follows:

Fixed. The structure and interpretation of data exchanged using the format is predetermined and not changeable.

Modifiable. The exchange format is based on flexible framework where the structure and interpretation of data can be modified. This is an important advantage for integrators because it eases the effort involved in synchronizing the structure for data that each tool will accept. It also makes it easier to add proprietary extensions to the exchange format as the need arises.

	Abstract Syntax						*Level of Abstraction*			*Type of Encoding*		*Transfer Mechanism*			*Schema Type*	
	Struct. Data	*Tree Only*	*Graph*													
			Typed	*Attrib.*	*Inher.*	*Hierar.*	*Low*	*Med*	*High*	*Text*	*Bin.*	*File*	*STS*	*DIF*	*Fixed*	*Mod.*
ASIS	■						■			■				■	■	
Aterms	■	□					■			■	■	■		■	■	
IML		■					■			■		■			■	
RG			■	■			□	■	■	■		■				■
CORUM						■	■			■				■	■	
CORUM II						■	■	■	■	■				■	■	
Datrix-TA			■	■			■			■		■			■	□
FAMIX	■						■	■		■			■			■
GraX			■	■	■		■	■	■	■			■			■
GXL			■	■	■	■	■	■	■	■			■			■
PROGRES			■	■	■	■	■	■	■	■		■				■
RSF			■	■			□	■	■	■		■			■	
TA			■	■	■		■	■	■	■		■				■
TA++			■	■	■		■	■		■		■			■	

Key:
■ = Designed for supporting the property indicated
□ = Capable of supporting the property indicated

Table 2: Characteristic Properties of Software Exchange Formats

6.2 Software Exchange Formats

Table 2 provides a summary of the characteristic properties of 14 formats for exchanging software representations that have been used in the reverse engineering field. Each of these exchange formats is described in more detail below.

6.2.1 ASIS

The *Ada95 Compilation Environment (ACE)* stores syntactic and semantic information about compiled Ada source code in a proprietary IR format. A linker makes use of this IR to create an executable Ada application. Although CASE tool and application developers can access information in ACE through the use of proprietary interfaces, they are difficult, time-consuming and expensive to develop. The *Ada Semantic Interface Specification (ASIS)* is an open source API written in Ada95 for accessing ACE information. It has been designed to be a portable means for developers to get syntactic and semantic information about Ada source code without being burdened with understanding the complexities of proprietary ACE internal representations. A number of types and subtypes are defined along with a set of operations that query the ACE and pass results back to the calling application [ASI 98a, ASI 98b, ASI 98c].

An example of ASIS in action from Rybin and Fofanov [RF 00] is shown below. In Figure 6 the package `Asis.Interator` is defined, providing procedures and operators that support the traversal of tree structures. In Figure 7 an instantiation of `Traverse_Element` from the `Asis.Interator` package is used as part of a tool operation that traverses the AST for Ada code checking for conformance to a specific coding style.

ASIS has been widely accepted within the Ada development community. The International Organization for Standardization ratified ASIS as an international standard in 1999 [ISO 99]. Dozens of code-level analysis tools have been built based on the ASIS API. Especially useful is

```
package Asis.Iterator is

   generic
      type State_Information is limited private;

      with procedure Pre_Operation
             (Element : in     Asis.Element;
              Control : in out Traverse_Control;
              State   : in out State_Information) is <>;

      with procedure Post_Operation
             (Element : in     Asis.Element;
              Control : in out Traverse_Control;
              State   : in out State_Information) is <>;

   procedure Traverse_Element
      (Element : in     Asis.Element;
       Control : in out Traverse_Control;
       State   : in out State_Information);

end Asis.Iterator;
```

Figure 6: The `Asis.Iterator` Package is Defined

```
Type Style_Check_State is (Not_Used);

procedure Check_Style_Rules is ... end Check_Style_Rules;

procedure No_Operation (...) is begin null; end No_Operation;

procedure Recursive_Style_Check is new
   Asis.Iterator.Traverse_Element
      (State_Information => Sytle_Check_State,
       Pre_Operation     => Check_Style_Rules,
       Post_Operation    => No_Operation);

procedure Process_Construct (Construct : Asis.Element) is
   Process_Control : Traverse_Control := Continue;
   Process_State   : Style_Check_State;
begin
   Recursive_Style_Check
   (Construct, Process_Control, Process_State);
end Process_Construct;
```

Figure 7: A Coding Style Checker that Recursively Traverses the AST for Ada Code

---

the ability for developers to create tools that "snap on" [CT 99] to Ada compilation systems such as the freely available *GNU NYU Ada 9X Translator (GNAT)* [RF 00, SB 94].

6.2.2 ATerms

Annotated Terms (ATerms) represent an exchange format and a library of API functions for manipulating them. The exchange format is designed to represent the data produced by parsers, structural editors, compilers and other components in software reengineering tools. The API was developed to encourage tool authors to integrate ATerms into their reengineering tools [BJO 00].

ATerms are constructed out of the following components [BKV 96a, BKO 98]:

- Integer and real numeric values.
- Function applications – A function symbol followed by 0 or more function arguments.
- Lists – Consisting of one or more ATerms.
- Tags – Each indicating an ATerm type.
- Binary Large data OBjects (BLOBs) – Binary data in ATerms format.
- Annotations – A list of label and corresponding annotation pairs, both in ATerms format.

Although only structured data is exchanged, ATerms are capable of representing tree structures such as ASTs [BJO 00].

A number of operations for working with ATerms are implemented in the API. The operations are organized into two categories. Thirteen Level 1 operations for creating, matching, reading, writing and annotating ATerms have been created. These are the standard operations most commonly used by tool developers. Level 2 operators provide advanced functionality for developers who, for some reason or another, need to access the core functionality of the API [BJO 00, BJK+ 00].

ATerms have a text and a binary format. The text format is designed to be readable and easily understood by humans. The main drawback of the text format is that it requires excessive storage space when being used to represent large systems. To get around this problem, a binary

Let the sets *C*, *N*, *L* and *F* be defined as follows:

$C = \{a, b, c\}$
$N = \{1, 2, 3\}$
$L = C \cup N$
$F = \{(f,1),(g,2),(h,3)\}$

Given these definitions some of the following types of ATerm instances can be constructed:

Constants:	`Abc`
Numerals:	`123`
Literals:	`"abc"` `"123"`

Lists:	`[]`
	`[1, "abc", 3]`
	`[1, 2, [3, 2], 1]`

Functions:	`f("a")`
	`g(1,[])`
	`h("1", f("2"), ["a","b"])`

Annotations:	`f("a"){g(2,["a","b"])}` `"1"{[1,2,3],"abc"}`

Figure 8: Example ATerms

format for ATerms called *Binary ATerms Format* (*BAF*) is available. Storing an ATerms node in memory requires 4.5 bytes while only 1.54 bytes are required to store the same node in BAF. The use of BAF also significantly reduces the time it takes to read and write to an ATerms file [BJK+ 00]. BAF is platform independent because it uses indexes rather than memory addresses to store and retrieve entities [BJO 00].

ATerms examples from [BKV 96a] are shown in Figure 8. ATerms are defined through the instantiation of a set of constants (*C*), a set of numerals (*N*), a set of literals (*L*) and a set of functions (*F*). The functions that make up set *F* consist of function symbols and a number that represents the number of parameters for the function (also known as the *arity* of the function). Square brackets are used to enclose lists. Curly brackets are used to enclose annotations.

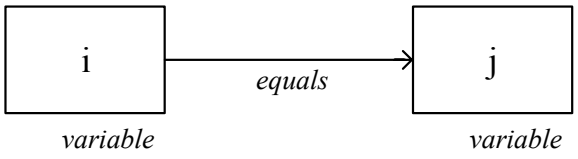
6.2.3 IML and RG

Bauhaus is a joint research project among the University of Stuttgart and the Fraunhofer Institut für Experimentelles Software Engineering in Kaiserslautern, Germany. The goal of the project is to build tools that help in the software reengineering process [Bau 01]. In particular, Bauhaus Group researchers are interested in providing a means for recovering architectural characteristics of software systems. To facilitate their distal research efforts, two formats for exchanging software representations among project participants have been developed [CEK+ 00].

The *InterMediate Language* (*IML*) is a portable IR that makes use of an attributed tree structure to represent details at the source code level. The semantics of the programming language constructs used in the code are preserved. Once the source code has been processed and represented in IML, advanced data-flow and control-flow analyses can be carried out. Although the format was not designed to be flexible, it is very efficient at representing software details at a low level of abstraction.

Unlike IML, the *Resource Graph* (*RG*) format represents source code at medium and high levels of abstraction. An E-R model is used to record information in a graph structure with entities and relationships respectively corresponding to nodes and edges on the graph. Entity and relationship instances are determined through direct source code examination and through a variety of analyses. For example, entities such as functions, types and variables are determined directly from the source code. Other entities such as abstract data types, components and subsystems are determined from analysis.

A simple schema is used to define the meaning for each node and edge type in the RG format. Instances are identified with unique labels and a corresponding type declared in the schema. Both the schema and the data being represented are stored in a single file. A graph with




```
NODE_TYPES
> variable NT1;
> variable NT2;
EDGE_TYPES
> equals ET1;
NODES
#1 i NT1 list-of-attributes ;
#2 j NT2 list-of-attributes ;
EDGES
#1 #2 ET1 list-of-attributes ;
```

Figure 9: A Simple Graph and It's Representation in RG Format

---

its corresponding representation in RG format is shown in Figure 9. This simple example is based on a modified version of the example provided in [CEK+ 00]. A graph visually represents the source code statement `i = j`. Two nodes of type `variable` and an edge of type `equals` are defined in the schema section at the beginning of the file. Node and edge instances are defined in the subsequent section. Note that an optional list of attributes (shown italicized) can be associated with each node and edge in the graph.

6.2.4 CORUM and CORUM II

Kazman et. al [KWC 98] describe the *Common Object-based Re-engineering Unified Model (CORUM)* as an API-based environment for integrating software reengineering tools. The original system outlined in [WOL+ 98] focuses on integrating tools that work at the source code level. Support for representing ASTs, symbol tables, Control Flow Graphs, Data Flow Graphs, Data Slices and Programming Plans is provided. A hierarchical format based on a Multi-Layer and Multi-Edge-Set (MLMES) graph [LCH+ 98] is used for all internal representations. On the MLMES graph each nodes corresponds to a node on the AST. Each edge records data and control flow information. Additional information is stored along with the graph as annotations.

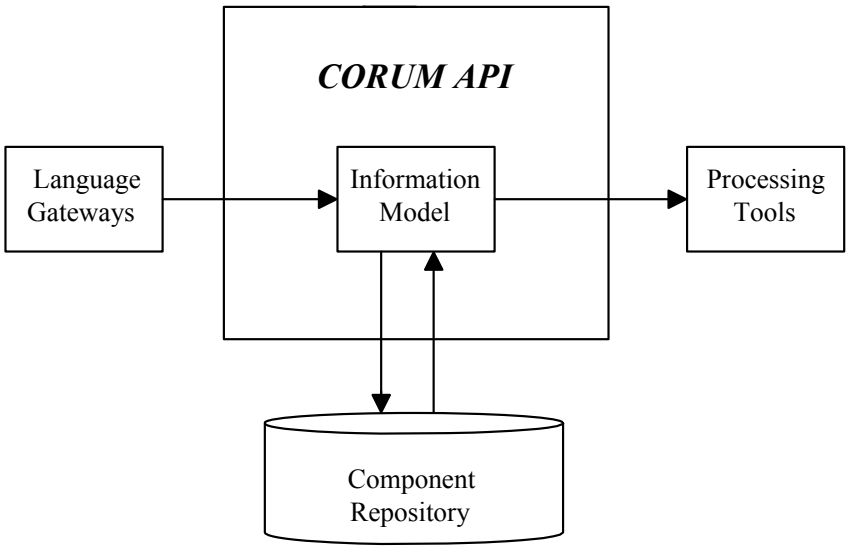


Figure 10: The CORUM Architecture

The architecture for CORUM is shown in Figure 10 [WOL+ 98]. The API functions as the primary integrator of language gateways that bring in code for analysis, a component repository for storing records and processing tools that provide analysis and end user access to information. The API itself is built around an information model that supports the many representations for software mentioned above.

The CORUM II framework is a proposal for enhancing CORUM to provide advanced functionality for analysis of a system at the architectural level of abstraction. This analysis includes identification of design patterns and coding styles. Architectural semantics are supported in CORUM II through temporal and static views of the system. Temporal views correspond to behavioural observations of the system that take place while the system is running such as:

- The time when the system accepts or relinquishes control.
- The time that the system accepts or transmits data.
- The new processes the system creates.

- The circumstances when the system preserves the state of a process.

Static views provide a functional perspective of the system based on information extracted from the source code such as:

- The scope for data and control.
- Changes that are made to data.
- Suspension of a process when control is transferred.
- Instances where control is given up voluntarily.
- Inputs and outputs for data and control.
- How many input or output elements can be connected to the same channel at the same time.

CORUM II supports the horseshoe software reengineering process outlined in Section 2.1.5 [KWC 98].

6.2.5 Datrix-TA

Bell Canada's Datrix system evaluates the maintainability of software. It can evaluate source code for reusability, track its evolution, perform rudimentary architectural analysis and characterize how clearly the code is written and structured. The system consists of Java, C and C++ parsers and linkers that take in the source code, analyzers that produce reports and an import/export module that facilitates exchange of information. All of these components are built around a central in-memory representation of source code called Datrix-ASG [Lap 00]. One of the advantages of the Datrix system is that the same ASG is used for representing Java, C and C++ source code.

Datrix-ASG is structured as a graph consisting of attributed nodes and edges. All nodes and edges have a type that fits into a hierarchy of ASG elements. The generic node type `<cAsgNds>`

is defined as an abstract class. Within this class there are abstract classes for identifiers, scopes, objects, functions, expressions, control statements and other classes. These classes are further subdivided into other classes and types. The same is true for edges. The `<cArc>` abstract class is subdivided into reference and syntax classes that are further subdivided into different types based on referential and syntactic relationships that can exist between nodes. Although the ASG is designed to be as language independent as possible, the format is flexible so that new nodes or edges can be added to support language specific concepts when necessary.

The representation currently has very little support for representing architectural components of the system being evaluated. One ASG element provides a means for considering scope in a global sense. A future goal is to provide elements that support the representation of modules, subsystems and other architectural components. Currently, obtaining a system-wide ASG requires the merging together of ASGs for each of the code sources that make up the system [LLL 01].

The ASG produced by the Datrix parsers is distinct from the *Abstract Syntax Tree* (*AST*) that parsers typically produce. The addition of semantic information introduces relationships that create circuits through the data structure yielding a strongly typed, attributed, directed graph. For example, in an AST an entity reference is represented by an edge that points to a terminal node in the tree structure. In the Datrix-ASG a subgraph is used for the declaration of an entity. This subgraph is shared among all instances of the entity. A reference to an entity is indicated as an edge that points to the top node of the declaration subgraph. Despite the graph structure, Datrix-ASG still holds a parser-produced AST for the source code within it. The additional semantic information simply decorates the AST, transforming the representation into a graph structure [LL 00].

An external format called *Datrix-TA* is used to exchange Datrix-ASGs among the different tools that make up the Datrix system. It is largely based on Holt's *Tuple Attribute (TA)* language (see Section 6.2.11) except that it has a fixed schema and some proprietary enhancements have been added to the language. Datrix-TA is a text-based format that stores information in a linear fashion without nested statements. This makes it easily readable by humans and efficiently parsed by computers. All nodes are uniquely identified. No naming or ordering restrictions for nodes have been established.

An example from [LL 00] is shown below. In Figure 11 a small C++ program called `alias.cpp` is shown. In this program a `typedef` is used to set up the identifier `foo` as a substitute for the `int` type. Two integers are declared, one using type `int` and one using type `foo`. The ASG for the code is shown in Figure 12. Although the graph is cluttered, it is easily understood. In a nutshell, the top node (`cScopeGlb`) refers to the global scope of the program. This is decomposed into the compilation scope (`cScopeCompil`) and the scope of the source file (`cScopeFile`), both of which are limited to the code that makes up the program. Two objects are instantiated: `x` is an instance of a `cBuiltInType` named `"int"` and `y` is an instance of a `cAliasType` called `"foo"` (which is also an instance of a `cBuiltInType` named `"int"`). The attributes `beg` and `end` show the line number and character position on the line where the entity being described is located. The `visb` attribute summarizes the visibility of each of the objects and alias that make up the program. All are `pub` meaning they are publicly visible within the scope of the program. It is interesting to note how much information is contained in the ASG for such a small program. The ratio of extracted information to code is very high.

```
// alias.cpp
typedef in foo;

int x;

foo y;
```

Figure 11: A Small C++ Program

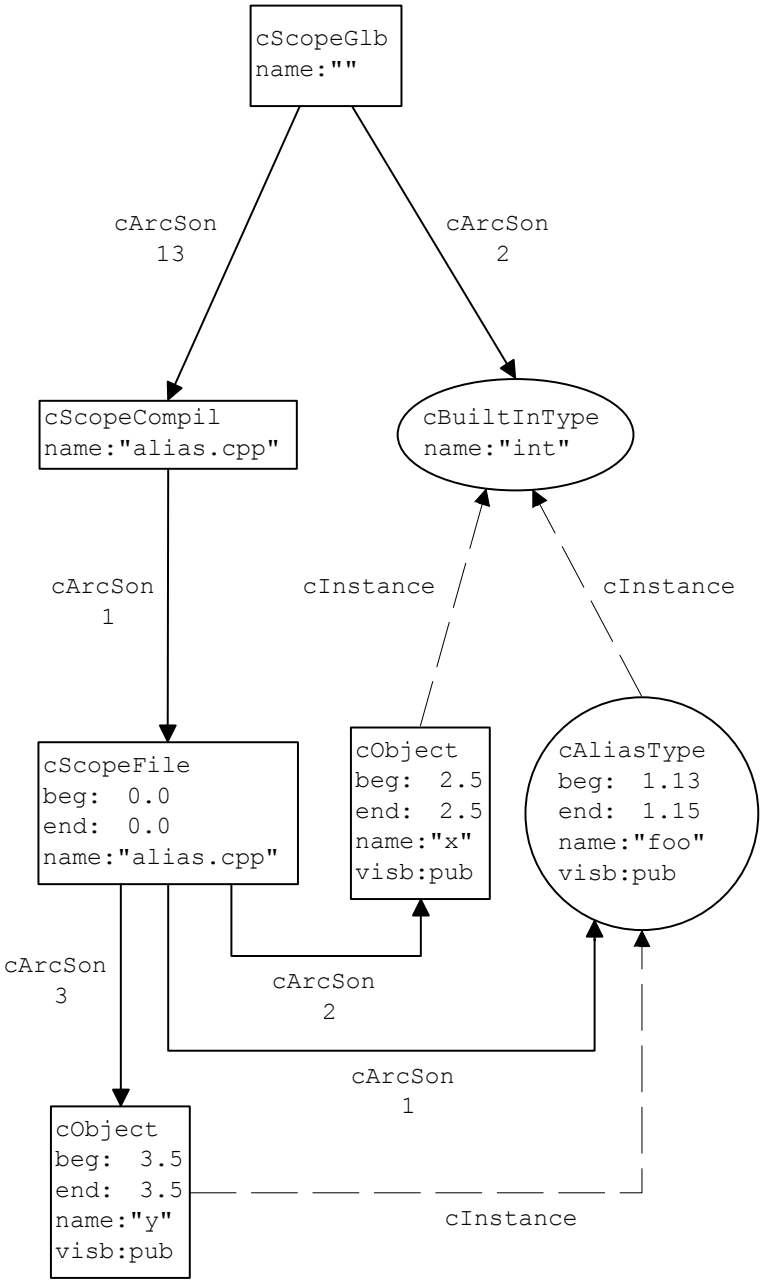


Figure 12: The ASG for the code in Figure 11

```
$INSTANCE 1 cAliasType
1{
beg = 1.13
end = 1.15
name = "foo"
visb = pub
}
$INSTANCE 2 cBuiltInType
2{
name = "unsigned char"
}
$INSTANCE 3 cBuiltInType
3{
name = "char"
}
$INSTANCE 4 cBuiltInType
4{
name = "float"
}
$INSTANCE 5 cBuiltInType
5{
name = "long double"
}
$INSTANCE 6 cBuiltInType
6{
name = "double"
}
$INSTANCE 7 cBuiltInType
7{
name = "unsigned long"
}
$INSTANCE 8 cBuiltInType
8{
name = "unsigned short"
}
$INSTANCE 9 cBuiltInType
9{
name = "unsigned int"
}
$INSTANCE 10 cBuiltInType
10{
name = "short"
}
$INSTANCE 11 cBuiltInType
11{
name = "long int"
}
$INSTANCE 12 cBuiltInType
12{
name = "int"
}
$INSTANCE 13 cBuiltInType
13{
name = "void"
}
$INSTANCE 14 cSystem
14{
name = ""
}
$INSTANCE 15 cScopeGlb
15{
name = ""
}
$INSTANCE 16 cScopeCompil
16{
name = "alias.cpp"
}
$INSTANCE 17 cScopeFile
17{
beg = 0.0
end = 0.0
name = "alias.cpp"
}
$INSTANCE 18 cObject
18{
beg = 2.5
end = 2.5
name = "x"
visb = pub
}
$INSTANCE 19 cObject
19{
beg = 3.5
end = 3.5
name = "y"
visb = pub
}
(cInstance 19 1)
{
}
(cArcSon 17 19)
{
order = 3
}
(cInstance 18 12)
{
}
(cArcSon 17 18)
{
order = 2
}
(cArcSon 17 1)
{
order = 1
}
(cInstance 1 12)
{
}
(cArcSon 16 17)
{
order = 1
}
(cArcSon 15 16)
{
order = 13
}
(cArcSon 15 2)
{
order = 12
}
(cArcSon 15 3)
{
order = 11
}
(cArcSon 15 4)
{
order = 10
}
(cArcSon 15 5)
{
order = 9
}
(cArcSon 15 6)
{
order = 8
}
(cArcSon 15 7)
{
order = 7
}
(cArcSon 15 8)
{
order = 6
}
(cArcSon 15 9)
{
order = 5
}
(cArcSon 15 10)
{
order = 4
}
(cArcSon 15 11)
{
order = 3
}
(cArcSon 15 12)
{
order = 2
}
(cArcSon 15 13)
{
order = 1
}
(cArcSon 14 15)
{
order = 1
}
```

Figure 13: The Datrix-TA Representation for the ASG in Figure 12

The ASG is represented in Datrix-TA format in Figure 13. The lengthy output has been compacted into three columns to save space. The graph nodes with their corresponding attributes are created first using `$INSTANCE` declarations. Next the edges between nodes and their appropriate orderings are defined.

6.2.6 FAMIX

The FAMOOS project at the University of Berne supports the reengineering of object-oriented software systems developed in Ada, C++, Java and Smalltalk. It provides an intermediate representation of object-oriented source code using the *FAMOOS Information Exchange Model (FAMIX)*. This representation is portable among a variety of reverse engineering tools from a variety of vendors. This is notable, since most exchange formats are developed to integrate among tools that exist within the same tool environment.

Demeyer, Ducasse and Tichelaar [DDT 99] argue against the use of UML for representing object-oriented source code in reverse engineering environments:

> "since UML is specifically targeted towards OOAD [Object-Oriented Analysis and Design], it lacks some concepts that are necessary in order to adequately model source-code … Of course it is possible to extend UML to incorporate these concepts, but then the protection of the standard is abandoned and with that the reliability necessary to achieve true interoperability."

FAMIX is presented as a superior alternative to UML. Its core model supports the representation of classes, methods and attributes. Within these entities, relations such as inherits, belongs-to, invokes and accesses can be established. A diagram of the core model is shown in Figure 14. The core model has been augmented to include other items such as functions, local variables, global variables and parameters. Each entity is given a unique name in the model, which facilitates the establishment of relationships. Unique names have other benefits as well. They eliminate the need to establish explicit relations that exist only in the transfer. This makes the information

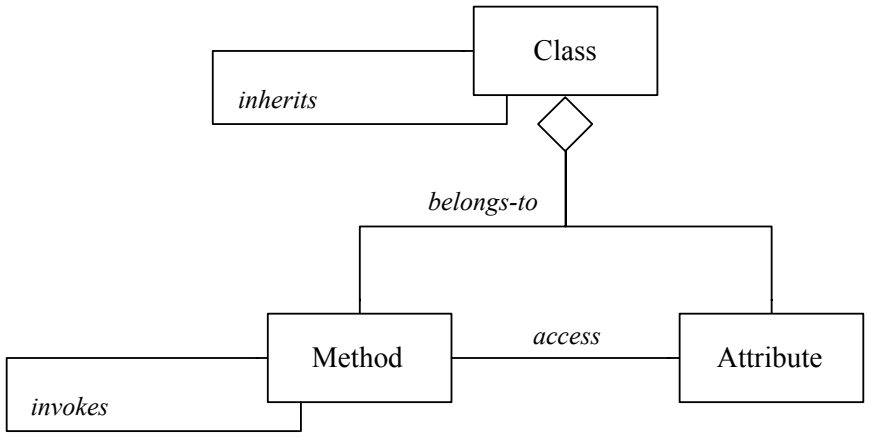


Figure 14: The Core Model of FAMIX

being transferred more human readable. It also benefits query functions that often involve the accumulation or merging of information from diverse sources [TDD 00].

Originally CDIF was used as the transfer mechanism for FAMIX structured data. Although CDIF was well suited for this purpose, the OMG and other reverse engineering tool vendors discontinued their support for it. Consequently, the FAMOOS project shifted its focus towards XMI for their transfer requirements. To make use of XMI, tool developers must first create a *Meta Object Facility (MOF)* compliant model that provides a structure for the information to be exchanged. Because of the object-oriented nature of FAMIX, the creation of such an MOF compliant model was not difficult. Following a number of standardized production rules, the MOF model is used to generate an XML DTD. Using this DTD, the structure of valid XMI files for exchanging FAMIX data is determined.

Other tool developers may not want to use XMI, especially those that provide representations for the significant quantity of legacy code that is not object-oriented. The FAMOOS project is interested in investigating possible mappings between FAMIX and UML. Since UML is also

```
package gui;

class Widget {
  int wTop;
     ...
  void print() {
    System.out.println("Top = " + wTop);
  }
}
```

Figure 15: A Small Extract of Java Code

```
 (Class FM1
 (name "Widget")
 (uniqueName "gui::Widget")
)
…
(Attribute FM5
 (name "wTop")
 (uniqueName "gui::Widget.wTop")
 (belongsToClass "gui::Widget")
)
…
(Method FM8
 (name "print")
 (uniqueName "gui::Widget.print()")
 (belongsToClass "gui::Widget")
)
…
(Access FM12
 (accesses "gui::Widget.wTop")
 (accessedIn "gui::Widget.print()")
)
```

Figure 16: FAMIX Representation For Java Extract From Figure 15 In CDIF

based on an MOF model, converting FAMIX to a MOF model was a logical first step towards this goal.

An example from [TDD 00] demonstrates the difference between FAMIX structured data in CDIF and XMI. In Figure 15 a small snippet of Java code is provided. The corresponding FAMIX representation for the code is shown in CDIF in Figure 16 and XMI in Figure 17. The model definitions and headers for each example have been excluded for clarity. CDIF is well laid out and easy to read. XMI is considerably more verbose and compact, and is much more challenging for humans to understand.

```
<Famix.Package xmi.id="_1">
 <Famix.Entity.name>gui</Famix.Entity.name>
 <Famix.Entity.uniqueName>gui</Famix.Entity.uniqueName>
</Famix.Package>
<Famix.Class xmi.id="_2">
 <Famix.Entity.name>Widget</Famix.Entity.name>
 <Famix.Entity.uniqueName>gui::Widget</Famix.Entity.uniqueName>
 <Famix.Class.belongsToPackage>gui</Famix.Class.belongsToPackage>
</Famix.Class>
<Famix.Method xmi.id="_3">
 <Famix.Entity.name>print</Famix.Entity.name>
 <Famix.Entity.uniqueName>gui::Widget.print()</Famix.Entity.uniqueName>
<Famix.BehaviouralEntity.signature>print()</Famix.BehaviouralEntity.signature>
 <Famix.Method.belongsToClass>gui::Widget</Famix.Method.belongsToClass>
</Famix.Method>
<Famix.Attribute xmi.id="_4">
 <Famix.Entity.name>wTop</Famix.Entity.name>
 <Famix.Entity.uniqueName>gui::Widget.wTop</Famix.Entity.uniqueName>
 <Famix.Attribute.belongsToClass>gui::Widget</Famix.Attribute.belongsToClass>
</Famix.Attribute>
<Famix.Access xmi.id="_5">
 <Famix.Access.accesses>gui::Widget.wTop</Famix.Access.accesses>
 <Famix.Access.accessedIn>gui::Widget.print()</Famix.Access.accessedIn>
</Famix.Access>
```

Figure 17: FAMIX Representation For Java Extract From Figure 15 In XMI

6.2.7 GraX

GraX [EKW 00, EKW 99] is used as a means for exchanging software representations between the *KOGGE Computer-Aided Software Engineering* (CASE) and the *GUPRO Computer-Aided Reengineering (CARE)* tools from the University of Koblenz-Landau in Germany. At the core of GraX is a graph model known as *TGraphs*. TGraphs have the following characteristics [EWD+ 96]:

- They are directed. One starting and ending vertex for each edge is defined.
- They are typed. Different classes are used to group edges and vertices. Multiple inheritance is supported using typing.
- They are attributed. Descriptive information can be associated with vertices and edges that make up a graph. The attributes that are allowed depend on the type for the vertex or edge that the attribute relates to. Attributes are recorded using pairs (tuples) that associate

a given attribute with the id for the vertex or edge. Attributes are not just simple strings. Structured information can also be stored as an attribute.

- Edges are 'first-class' entities. That is to say, edges are equal to all other entities with respect to their individual properties.
- They are ordered. All the edges that are incident to a particular vertex are numbered, providing a means for articulating entity sequences. This numbering is distinct from the edge identifier, which is presumably also a number.

Note that the above-mentioned characteristics are all optional. They may or may not be present in a TGraph. For example, TGraphs can be used to represent undirected, typed, attributed, unordered graphs.

One feature of TGraphs is that they are very general. They can be used to represent trees, DAG-like graphs, undirected, relational graphs and many other types of graphs useful in the reengineering tool domain. TGraphs are capable of representing software at various levels of abstraction.

A number of examples are provided in [EKW 99]. A TGraph representation of the ASG for the code fragment in Figure 18 is shown in Figure 19. Each of the edges and vertices have a unique identifier and type. Vertices `v3`, `v4`, `v5`, `v8`, `v10`, `v11`, `v13` and `v16` have additional attributes where values or identifiers are instantiated. The four outgoing edges on vertex `v3` are ordered to indicate four instances where variable x is referenced.

```
while x > 0 do
   repeat
     y := y + 1
   until (y = x);
   x := x _ 1
od
```

Figure 18: A Fragment of Source Code

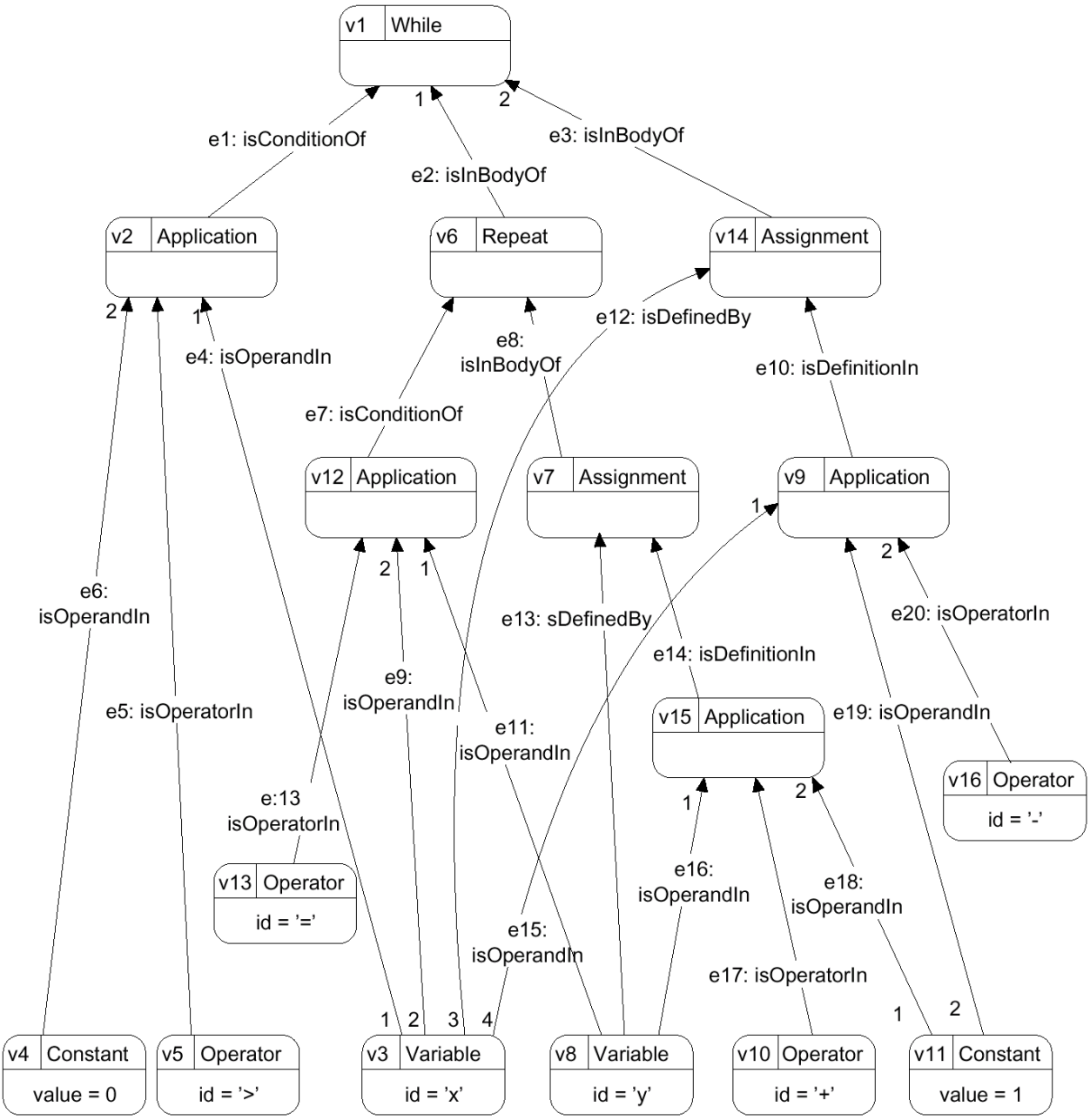


Figure 19: TGraph of the ASG for the Source Code Fragment in Figure 18 [EKW 99]

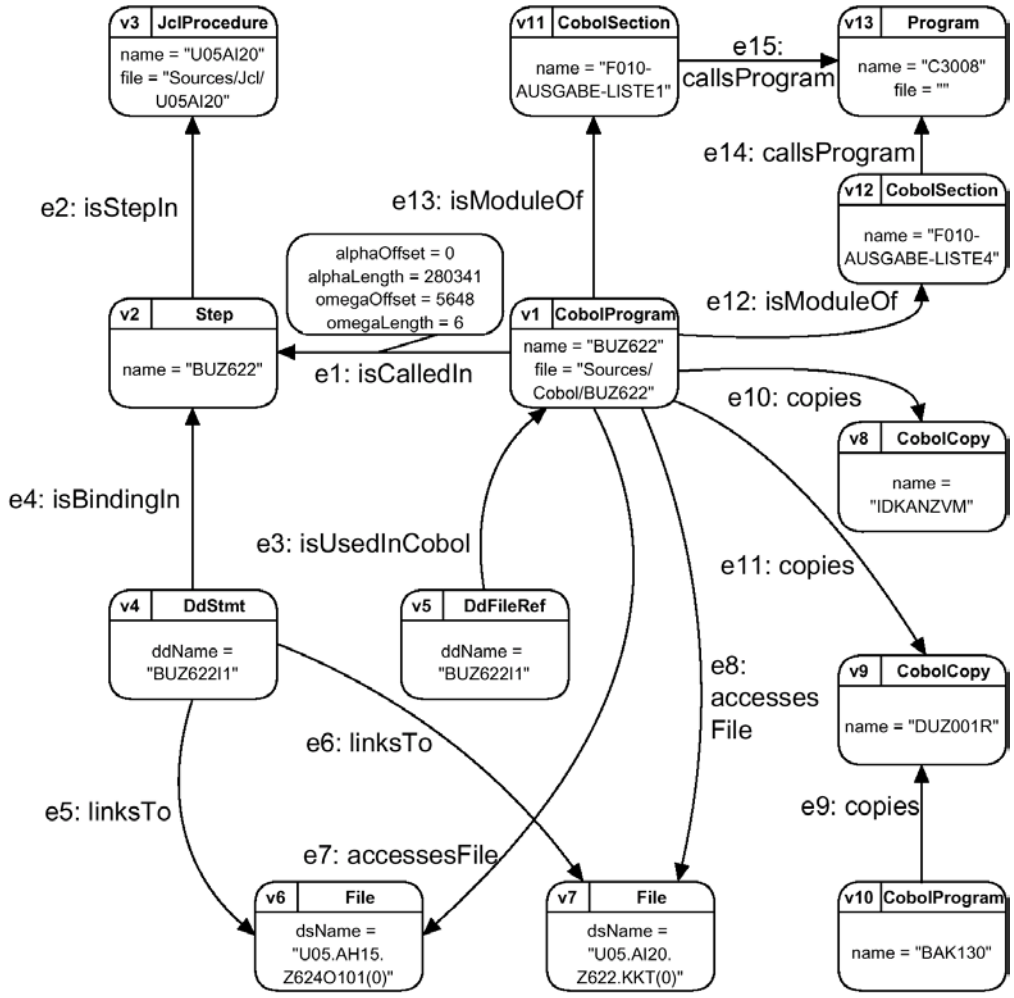


Figure 20: Extract of a TGraph for a Large Multi-Language System [EKW 99]

In another example, an extract of a coarse-grained TGraph representation for a multi-language system in a large insurance company is shown (Figure 20). Extensive use is made of TGraph's ability to store attributes for vertices and edges. For instance, edge `e1` has an attribute that provides information on the `isCalledin` relationship between vertices `v1` and `v2`. These examples demonstrate how effective TGraphs are for fine-grained representation of source code to coarse-grained architectural modeling of multi-language systems.

GraX exchanges TGraph representations using XML as a wrapper and transport mechanism. One nice feature of TGraphs is that they can be used to represent schema information and data

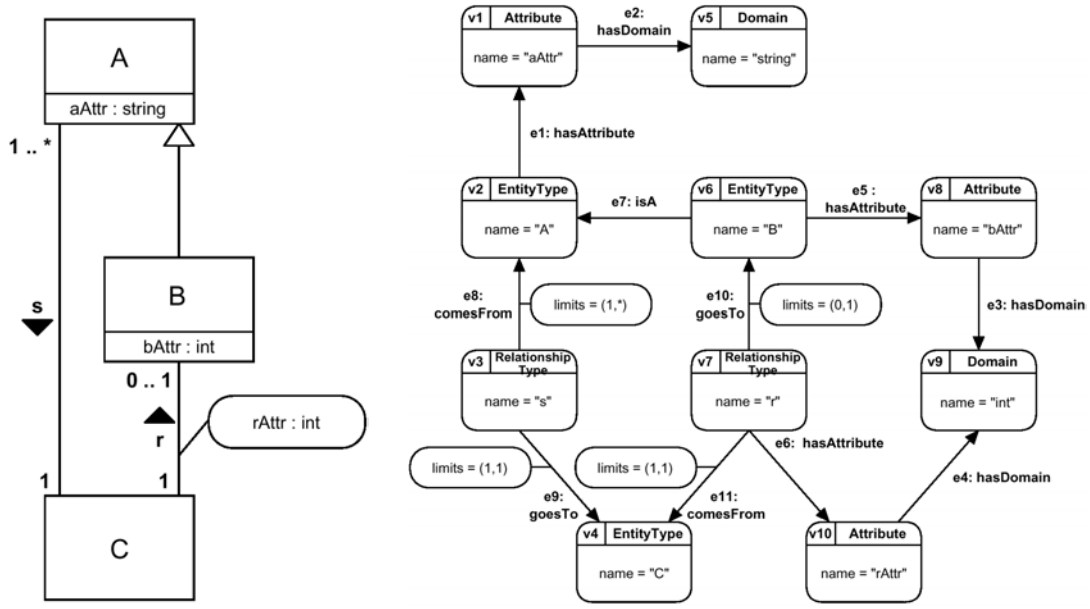


Figure 21: A Conceptual Model for Attributed Graphs and Its TGraph Representation

information. In this way, the XML DTD only needs to define TGraphs. Once the DTD is in place, reengineering tools can exchange schema information and data information as TGraphs using XML. An example of this capability from [EKW 00] is shown. In Figure 21 the conceptual model for a class of attributed graphs is shown on the left. An equivalent representation of the model is shown as a TGraph on the right. The corresponding GraX document for the model is shown in Figure 22. This GraX document acts as a schema for a class of attributed graphs that can subsequently be represented as TGraphs and exchanged as GraX documents.

6.2.8 GXL

The *Graph Exchange Language (GXL)* [HW 00a, HW 00b] is designed to exchange software representations at all levels of abstraction. It takes advantage of many of the characteristics found in other popular exchange formats. In particular, the exchange formats *GraX*, *PROGRES*, *RSF* and *TA* as well as a means for manipulating them (*Relational Partition Algebra* [FV 99, Kri 97]), have all contributed significantly to GXL. Holt, Winter and Schürr [HWS 00] list 42 instances where TA, GraX, PROGRES and RSF have been used facilitate tool interoperability and support

```
<?xml version="1.0" ?>
<!DOCTYPE grax SYSTEM "grax.1.0.dtd" >
<grax schema = "meta.1.0.scx" >
    <vertex id = "v1" type = "Attribute" >
        <attr name = "name" value = "aAttr"/>
    </vertex>
    <vertex id = "v2" type = "EntityType" >
        <attr name = "name" value = "A"/>
    </vertex>
    <vertex id = "v3" type = "RelationshipType" >
        <attr name = "name" value = "s"/>
    </vertex>
    ...
    <edge id = "e7" type = "isA"
        alpha = "v6" omega = "v2" >
    </edge>
    <edge id = "e8" type = "comesFrom"
        alpha = "v3" omega = "v2" >
        <attr name = "limits" value = "(1,*)" />
    </edge>
    <edge id = "e9" type = "goesTo"
        alpha = "v3" omega = "v4" >
        <attr name = "limits" value = "(1,1)" />
    </edge>
    ...
</grax>
```

Figure 22: The GraX Document Used To Exchange the TGraph in Figure 21

---

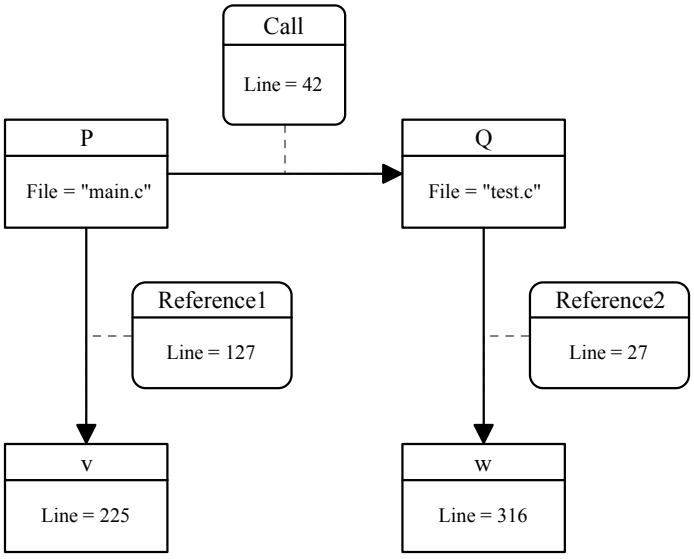
the reverse engineering of large-scale software systems. Drawing on this successful legacy, GXL is a scaleable and flexible integrator for reverse engineering tools operating on large sized industrial software systems.

GXL came about from extensive discussions among members of the reverse engineering community [EKW 01, Hol 98b, Hol 98c, Mül 98, Sch 00, SHK 00b, Sim 00c]. As a result, the format incorporates many features that are a direct benefit to reverse engineering practitioners. GXL represents attributed, typed, directed and undirected graphs in a straightforward and compact format. Hypergraphs, graphs with ordered edges and hierarchical graphs are also supported. Provision is made for storing complex values for attributes. GXL is designed to be extensible so that new graph elements or attribute types can be added [Win 01].

GXL uses XML as the transport mechanism for information exchange. The GXL DTD defines the syntactical structure of the information that will be passed between participating tools. At both ends of a transmission the GXL DTD can be used to check the validity of the information to be sent or received. Many checkers, parsers and other tools that work with XML documents are available. This makes it easier to integrate GXL exchange functionality into existing reverse engineering tools. Converters are also available for data stored in PROGRES, RSF and TA formats [HWS+ 00, HW 00c].

The representational backbone of GXL is based on GraX's TGraphs. TGraphs provide a means for ordering the nodes and edges that are incident to a given node. This feature is useful when modeling ordered relationships or graphing algorithms. The generality of TGraphs allows them to be used for all kinds of software representations. Conceptual models of schemas can be represented as typed graphs using TGraphs. Like GraX, GXL exchanges both schema and instance data using the same XML channel. A number of tools handle typed graphs in general. These tools can be 'oriented' to deal with specific kinds of typed graphs through the exchange of the GXL DTD, schema information and appropriately structured instance data.

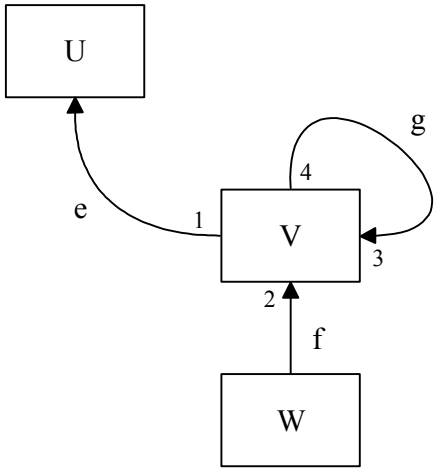
GXL does have some shortcomings. XML adds a lot of overhead in the form of tags to the information being exchanged. Reverse engineering domains are typically concerned with very large-scale systems. As a consequence, GXL formatted XML documents are very large. The notational overhead makes it difficult to process GXL-encoded data efficiently. Access to the vast quantity of information involved in reengineering analyses would be better achieved using an API accessing a repository. GXL is better suited as an integration enabler, providing an XML stream for exchanging software representations between tools.




```
<gxl>
<graph>
    <node id = "P" >
        <attr name = "File">
        <string> main.c </string>
        </attr>
    </node>
    <node id = "Q" >
        <attr name = "File">
        <string> test.c </string>
        </attr>
    </node>
    <node id = "v" >
        <attr name = "Line">
        <int> 225 </int>
        </attr>
    </node>
    <node id = "w" >
        <attr name = "Line">
        <int> 316 </int>
        </attr>
    </node>
    <edge id = "Reference1"
        from = "P" to = "v">
        <attr name = "Line">
        <int> 127 </int>
        </attr>
    </edge>
    <edge id = "Reference2"
        from = "Q" to = "w">
        <attr name = "Line">
        <int> 27 </int>
        </attr>
    </edge>
    <edge id = "Call"
        from = "P" to = "Q">
        <attr name = "Line">
        <int> 316 </int>
        </attr>
    </edge>
</graph>
</gxl>
```

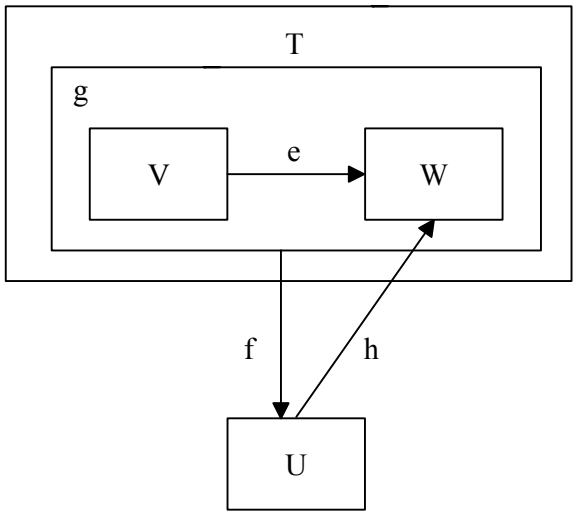
Figure 23: A Directed Graph Represented in GXL

A number of graphs and their corresponding representation in GXL reproduced from [Win 01] are shown. In Figure 23 a directed graph shows the call and reference relations among two procedures (`P` and `Q`) and two variables (`v` and `w`) in a hypothetical software system. Dashed lines are used to associate relation attributes to their corresponding relations (shown as solid, directed lines). Note that node and edge attributes are not simply stored as string phrases. In this




```
<node id = "U" />
<node id = "V" />
<node id = "W" />
<edge id = "e"
    from = "V" to = "U"
    fromorder = "1" />
<edge id = "f"
    from = "W" to = "V"
    toorder = "2" />
<edge id = "g"
    from = "V" to = "V"
    toorder = "3"
    fromorder = "4" />
```

Figure 24: A Directed Graph with Ordered Edges Represented in GXL




```
<node  id = "T">
    <graph id = "g">
        <node  id = "V"/>
        <node  id = "W"/>
        <edge  id = "e"
            from = "V" to = "W"/>
    </graph>
</node>
<node  id = "U"/>
<edge  id = "f"
       from = "T" to = "U"/>
<edge id = "h"
      from = "U" to = "W"/>
```

Figure 25: A Hierarchical Graph Represented in GXL

---

example, a name and a corresponding string or integer value are defined for each attribute. Any number of attributes can be defined for an entity in GXL in this manner.

In Figure 24 a directed graph with ordered edges is shown. The edge attributes `fromorder` and `toorder` provide a means for ordering edges, even when the source and destination node are the same (as is the case for edge `g`).

Hierarchical graphs are easily represented in GXL. Figure 25 provides an example. Node `T` consists of a directed graph `g`. Within graph `g` the nodes `V` and `W` are joined by the directed edge `e`. Although node `U` is instantiated adjacent to node `T`, edges `f` and `h` establish a hierarchical relationship to nodes `T` and `W`. This simple example demonstrates how GXL can be used to represent complex structures in a compact format. Despite the verbosity of the XML tags, only 10 lines were required to create this 3-level hierarchical graph.

6.2.9 PROGRES

The *PROgramming with Graph Rewriting Systems* (*PROGRES*) [Sch 97a, Sch 97b] environment consists of an integrated set of freeware tools that help developers create, analyze, compile and debug specifications for graph rewriting systems. Some of the significant tools in the environment are:

- **Syntax-Directed Editor**. Developers can create graph specifications that conform to the context-free syntax for the language they are working with.
- **Analyzer**. This tool performs an evaluative pass on a specification, detecting many types specification errors and identifying possible typographical errors.
- **Interpreter**. A tool used for validating specifications interactively. Productions based on the context-free syntax for the language can be implemented, checked and repaired. Additional functionality includes support for tracing, replaying and undoing changes to a specification.
- **Compiler Backends**. These tools generate code in C and Modula-2 that can be integrated into graph rewriting systems.

The representational backbone of PROGRES is based on directed, attributed graphs. Types for nodes and directed edges are established using labels. Node instances take on the properties

defined by the type that the node belongs to. Based on their type, edge instances are restricted to specific node types for their source and destination. Attributes can be associated only with nodes. At a higher level of semantics, different types of node types – referred to as *node classes* – can be established. This notion is useful for organizing common properties among different node types. PROGRES graphs are stored in a proprietary repository system called GRAS [JSZ 96, KSW 95].

An example of a PROGRES specification for a list data structure reproduced from [AE 95] is shown in Figure 26. At the beginning of the specification in the `Graph_Scheme` section all the allowable node and edge entities are defined. In the subsequent section called `Productions` all allowable operations on the node and edge entities with appropriate restrictions to their behaviour are outlined. The third section of the specification is shown in Figure 27, where a program (referred to as a `transaction`) makes use of the entities and productions defined to create a list structure consisting of three nodes.

PROGRES has been used to support the development of integrated tools for developing and maintaining large-scale software systems. For instance, the *Integrated Project Support Environment (IPSEN)* [Nag 96] uses graphs extensively as a foundation for all activities. Formal specifications represented as graph grammars are processed to create graph-based configurations for the system. Research into the recovery and transformation of software architecture representations by Fahmy, Holt and Cordy [FHC 01, FH 00a, FH 00b] has greatly benefited from the graph manipulation features of PROGRES.

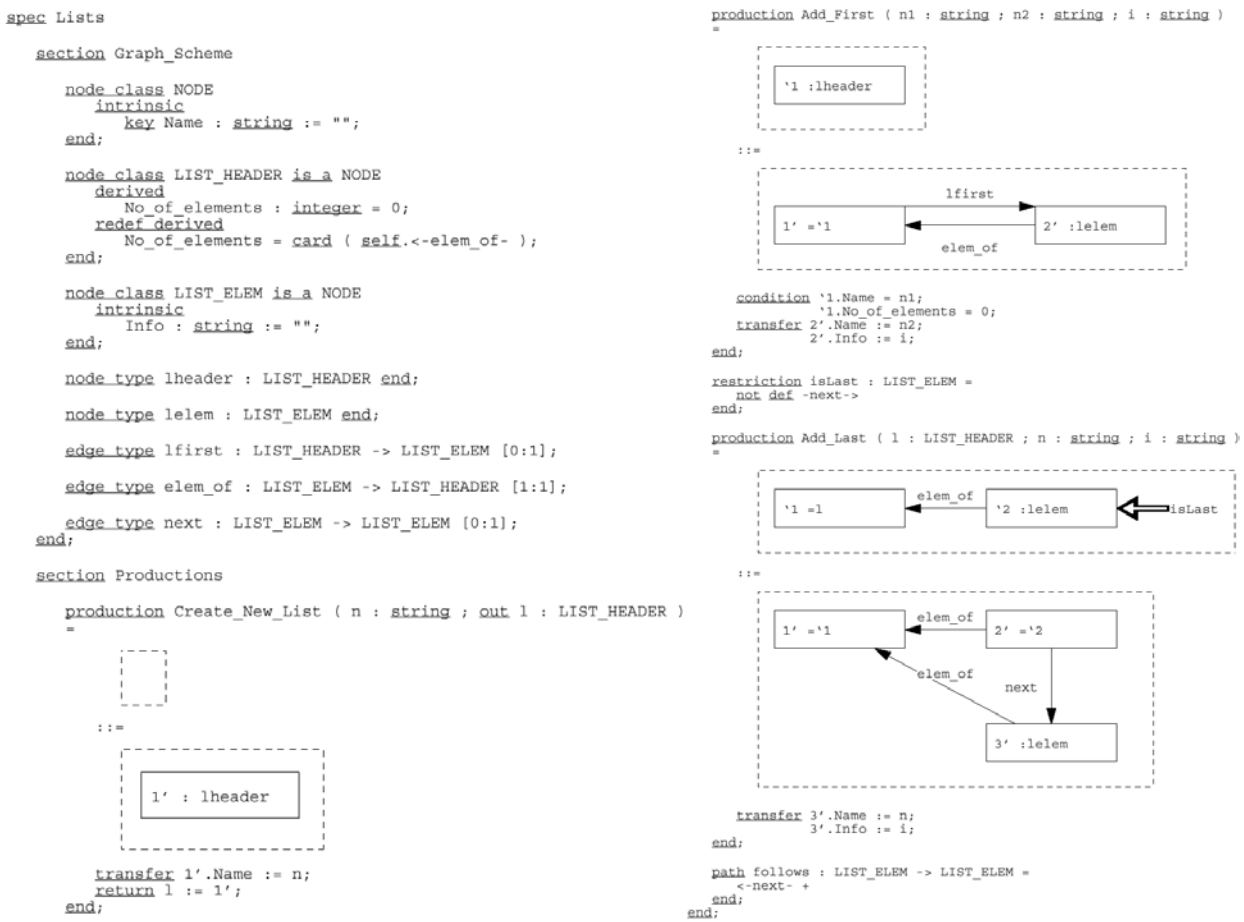


Figure 26: A PROGRES Specification For A List Structure [AE 95]

```
transaction MAIN =
   use l1 : LIST_HEADER
   do
        Create_New_List ( "list1", out l1 )
      & Add_First ( "list1", "element1", "short" )
      & Add_Last ( l1, "element2", "list" )
   end;
```

Figure 27: A PROGRES Transaction Creates A List Structure With 3 Nodes [AE 95]

6.2.10 RSF

Rigi [Won 98, MOT+ 93] is a repository-based modeling tool that supports the analysis of programs and documents from a reverse engineering perspective. It assists in process of extracting, organizing and abstracting information from source documents. Output from the tool is displayed in a graphical format for users. The main component of Rigi is `rigiedit`, a graph editor that supports the display, modification and analysis of graphical models produced from

documents being analyzed. Other components in Rigi include parsers for C, C++ and Java and tools for processing, sorting and storing information extracted from document artifacts.

Rigi is flexible, extensible and scalable. Different views of the software system being studied are supported. These views are interactive and provide output in an organized fashion similar to database views. A powerful scripting language provides the user with the means for automating and customizing different analyses. Besides built-in operations, other external algorithms can also be utilized for clustering, pattern matching, graph layout, slicing and other analyses. Rigi has been used successfully within an industrial setting to analyze systems with large source code bases (57,000 and 82,000 lines of code).

Integration among the components that make up Rigi is facilitated by the *Rigi Standard Format (RSF)*. Although the name implies a standard format, there are actually four different flavours of RSF [Mar 99]: *Unstructured RSF*, *Partly Structured RSF*, *Structured RSF* and *4-Tuple Unstructured RSF*. At its core, RSF is a simple, text-based interchange format that stores typed graphs with attributes and relations along with layout options for color. Nodes are listed by name. Edges and attributes are constructed using tuples. Small structural differences distinguish each of the flavours. According to Martin [Mar 99] all of the RSF flavours are inadequate because they do not support the storage of domain information and they do not handle double quotes in strings correctly. Unstructured RSF and 4-Tuple Unstructured RSF in particular do not support attributes for edges. A fifth RSF flavour is proposed that would supercede 4-Tuple Unstructured RSF and resolve these shortcomings.

Within the context of software architecture transformation, Fahmy, Holt and Cordy [FHC 01] provide a small example of RSF in use as a means for exchanging information among various tools that make up the *Portable Bookshelf (PBS) Toolkit* [PBS 01, FHK+ 97, Hol 97]. The graph

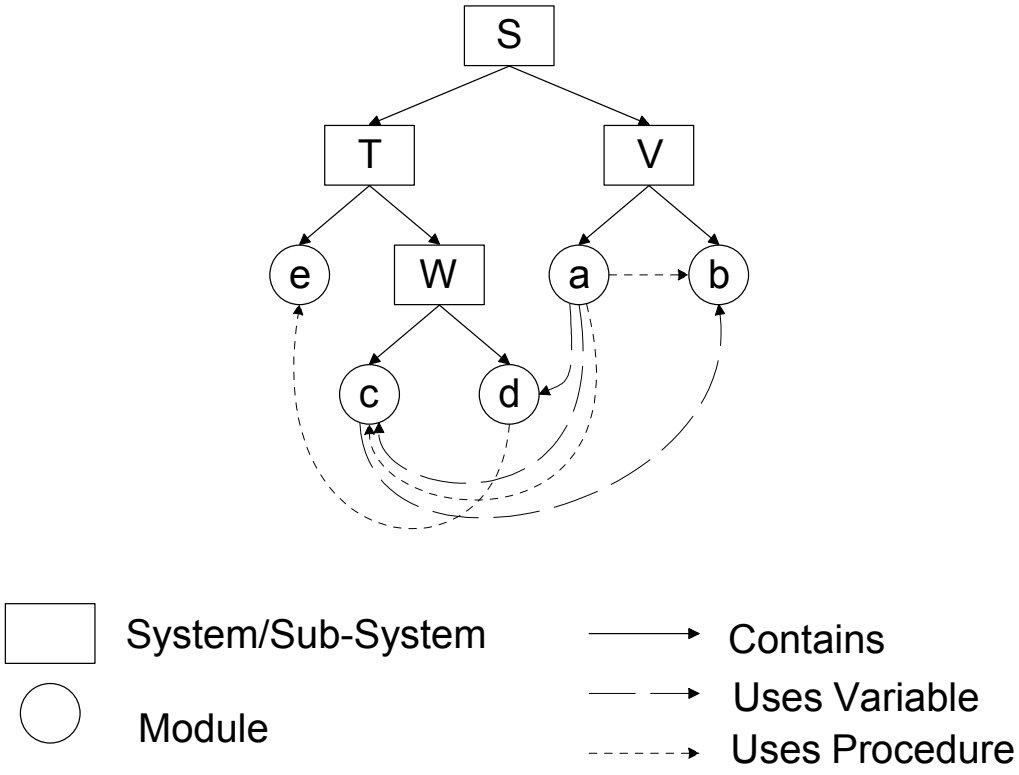


Figure 28: The Architecture of System

---

in Figure 28 shows the architecture of a sample system `s`. The graph consists of two types of entities: systems/subsystems (shown as rectangles) and modules (shown as circles). Three directed relationships among the entities are defined: contains (shown as a solid arrow), uses a variable (shown as a broken arrow) and uses a procedure (shown as a dotted arrow). The information contained within the graph can be expressed in terms of sets as shown in Figure 29. The same information is shown in Unstructured RSF in Figure 30. This token example shows the simplicity and compactness of RSF. There is no need to predefine entities. They are established 'on the fly' as the each of the relationships is declared.

Nodes = {*S*, *T*, *V*, *W*, *a*, *b*, *c*, *d*, *e*}
Contains = {(*S*, *T*), (*S*, *V*), (*T*, *e*), (*T*, *W*), (*W*, *c*), (*W*, *d*), (*V*, *a*), (*V*, *b*)}
UsesVariable = {(*a*, *d*), (*c*, *b*), (*a*, *c*)}
UsesProcedure = {(*a*, *b*), (*a*, *c*), (*d*, *e*)}

Figure 29: The Information In Figure 28 Expressed As Sets of Entities and Relations

```
Contains S T
Contains S V
Contains T W
Contains T e
Contains W c
Contains W d
Contains V a
Contains V b
UsesVariable a d
UsesVariable a c
UsesVariable c b
UsesProcedure a b
UsesProcedure a c
UsesProcedure d e
```

Figure 30: The Information In Figure 28 Expressed As Unstructured RSF

6.2.11 TA

The *Tuple-Attribute (TA)* language [Hol 98a] expresses graphs that represent large software programs. The TA language is divided both horizontally and vertically. Horizontally a distinction is made between *tuples* and *attributes*. Tuples are pairs that are used to define entities and relationships. Entities are represented as graph nodes while relationships are represented as graph edges. Vertically, TA is divided into two sub-language levels. The lower language level records instance information that is essentially the graph data. Holt refers to this as the "facts" level. The upper language level is used to define the structure for each type of entity and relation that is allowed in the graph. Holt refers to this as the "scheme" level. The scheme essentially describes the E-R diagram for graphs that can be fashioned from the structural characteristics provided.

Table 3 shows the four sections that make up the TA format. They are summarized using Holt's terminology as follows:

Language Level	Entity Type: *Tuple*	Entity Type: *Attribute*
Scheme	Scheme Tuple	Scheme Attribute
Fact	Fact Tuple	Fact Attribute

Table 3: Sections of the TA Format

1. **Scheme Tuple**. This section defines the structural characteristics for all entities and relations that can exist between them.
2. **Scheme Attribute**. This section defines the structural characteristics for all attributes that are used to describe the entities and relations defined in the Scheme Tuple.
3. **Fact Tuple**. This is the "data" section, where all entities and relations are instantiated.
4. **Attribute Tuple**. Attributes for the entity and relation instances in the Fact Tuple section are recorded.

The TA language is simple yet powerful. The text-based entries are easy to read and understand. Inheritance is supported at the scheme level as a property for entities and relations [EKW 99] using a special `$INHERIT` relation label. Nesting of attributes can also be set up at the scheme level. Given a graph, its conformance to a given scheme can be checked. The scheme in combination with the facts can be thought of as a graphical database for the software it represents.

An example based on one provided in [Hol 98a] is shown below. The graph in Figure 31 shows the call and reference relations among procedures `P` and `Q` and variable `V`. In Figure 32 the graph is represented in TA format. Line numbering has been added to the example to make it easier to explain the different parts of the TA file.

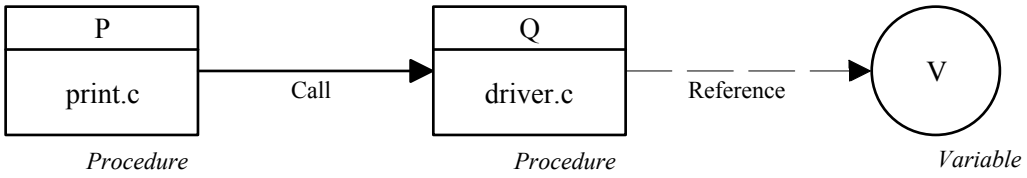


Figure 31: A Small Call/Reference Graph

```
1   SCHEME TUPLE :
2   Call Procedure Procedure
3   Reference Procedure Variable
4   SCHEME ATTRIBUTE :
5   Procedure { x y shape description }
6   Variable { x y shape }
7   Call   { line }
8   Reference { line }
9   FACT TUPLE :
10  $INSTANCE P Procedure
11  $INSTANCE Q Procedure
12  $INSTANCE V Variable
13  Call P Q
14  Reference Q V
15  FACT ATTRIBUTE :
16  P { shape = rectangle x = 50 y = 100 }
17  Q { shape = rectangle x = 150 y = 100 }
18  V { shape = "sm circle" x = 250 y = 100 }
19  P { description = print.c }
20  Q [ description = driver.c }
21  (Call P Q) { line = solid }
22  (Reference Q V) { line = dashed }
```

Figure 32: The Graph in Figure 31 Represented in TA Format

In the `SCHEME TUPLE` section two relations among procedure and variable entities are established. The name for the relation is stated first followed by the pre- and post-fixated entity arguments. So on lines 2 and 3 the statements "procedures call procedures" and "procedures reference variables" are respectively being defined. In the `SCHEME ATTRIBUTE` section various attributes for entities and relations are defined. For example, all `Procedure` entities have (`x`, `y`) coordinates, a `shape` and a `description` attribute associated with them (line 5).

In the `FACT TUPLE` and `FACT ATTRIBUTE` sections the entities, relations and their associated attributes are instantiated. Only the entities, relations and attributes defined in the `SCHEME TUPLE`

and `SCHEME ATTRIBUTE` sections are allowable. Entities are instantiated using the `$INSTANCE` label in lines 10, 11 and 12. The `Call` and `Reference` relations are instantiated in lines 13 and 14. On lines 16 to 22 various attributes for the entities are set. Only those attributes defined in the `SCHEME ATTRIBUTE` section are allowed. TA is flexible in its handling of text labels. Single words are accepted as they are. Phrases consisting of words separated by spaces are accepted if they are enclosed in double quotes. This is demonstrated on line 18 where the `sm circle` shape attribute for variable `v` is set.

6.2.12 TA++

The goal of the *Knowledge-Based Reverse Engineering of Legacy Telecommunications Software (KBRE)* project at the University of Ottawa is to assist software engineers in the exploration and modification of large complex software systems. To this end, they have developed a graphical code-browsing tool called *TkSee* that works with telecommunications system code written in Pascal at Mitel Corporation. A modified version of TA is used for representing and manipulating software representations among components that make up the tool. The format is referred to as TA++. The only major difference between TA and TA++ is the implementation of a fixed schema for TA++. This simply means that the `SCHEME TUPLE` and `SCHEME ATTRIBUTE` sections of TA++ are not modifiable.

TA++ makes extensive use of the inheritance capabilities of TA to create a hierarchy of entity classes used in TkSee. The complete scheme for TA++ [Let 98] is shown in Figure 33. The allowable entities and relations show that TA++ is suitable for representing software at the code level and at a medium level of abstraction.

```
SCHEME TUPLE :
$INHERIT SoftwareObject $ENTITY
$INHERIT SourceUnit SoftwareObject
$INHERIT Definition SoftwareObject
$INHERIT ReferenceExistence SoftwareObject
$INHERIT Subsystem SoftwareObject
$INHERIT SourceFile SourceUnit
$INHERIT SourceWithinFile SourceUnit
$INHERIT RoutineSource SourceWithinFile
$INHERIT ClassSource SourceWithinFile
$INHERIT TypedDefinition Definition
$INHERIT EnumerationConst Definition
$INHERIT StandaloneDefinition TypedDefinition
$INHERIT Field TypedDefinition
$INHERIT TypeDef StandaloneDefinition
$INHERIT DatumDef StandaloneDefinition
$INHERIT RecordTypeDef TypeDef
$INHERIT EnumeratedTypeDef TypeDef
$INHERIT CommentTermExistence ReferenceExistence
$INHERIT FileInclusionExistence ReferenceExistence
$INHERIT DataUseExistence ReferenceExistence
$INHERIT RoutineCallExistence ReferenceExistence
$INHERIT ManifestConstExistence DataUseExistence
$INHERIT TypeUseExistence DataUseExistence
potentiallyIncludedIn FileInclusionExistence SourceUnit
definedBy StandaloneDefinition SourceUnit
containingSource SourceWithinFile SourceUnit
declaredAsFormalArgsIn DatumDef RoutineSource
returnType RoutineSource TypeUseExistence
potentiallyCalledBy RoutineCallExistence RoutineSource
foundInSource CommentTermExistence SourceUnit
usedInSource DataUseExistence SourceUnit
ofType TypedDefinition TypeUseExistence
isEnumerationMemberOf EnumerationConst EnumeratedTypeDef
isFieldMemberOf Field RecordTypeDef
isMemberOf SourceFile Subsystem

SCHEME ATTRIBUTE :
SourceFile { path version dateChanged size }
SourceWithinFile { startChar endChar }
RoutineSource { isClassMethod, visibility }
StandaloneDefinition { startChar endChar }
DatumDef { isConst }
```

Figure 33: The TA++ Scheme Definitions

TA++ uses an interesting convention for identifying objects. A unique identifier is followed by an exclamation mark (‘!’) and then the name of the object. This helps to distinguish multiple objects with the same name in a software representation. Although the format is more difficult to read, it is more efficient for computer processing. In Figure 34, a description for the source code

```
FACT TUPLE:
$INSTANCE 0!ftfmctab.asm SourceFile
$INSTANCE 81!ftfmctab SpecialCodeExistence
foundInSource 81!ftfmctab 0!ftfmctab.asm
$INSTANCE 171!tabmac.inc FileInclusionExistence
potentiallyIncludedIn 171!tabmac.inc 0!ftfmctab.asm
$INSTANCE 186!list RoutineCallExistence
potentiallyCalledBy 186!list 0!ftfmctab.asm
$INSTANCE 193!ftfmctab SpecialCodeExistence
foundInSource 193!ftfmctab 0!ftfmctab.asm
$INSTANCE 261!ftf_tbl DataUseExistence
usedInSource 261!ftf_tbl 0!ftfmctab.asm
$INSTANCE 253!tabstrt RoutineCallExistence
potentiallyCalledBy 253!tabstrt 0!ftfmctab.asm
$INSTANCE 272!tabadd RoutineCallExistence
potentiallyCalledBy 272!tabadd 0!ftfmctab.asm
$INSTANCE 292!tstart RoutineCallExistence
potentiallyCalledBy 292!tstart 0!ftfmctab.asm
$INSTANCE 307!chartab RoutineCallExistence
potentiallyCalledBy 307!chartab 0!ftfmctab.asm
$INSTANCE 231!ftf_tbl DatumDef
definedBy 231!ftf_tbl 0!ftfmctab.asm
$INSTANCE 231!ASSEMBLER_TYPE TypeUseExistence
type 231!ftf_tbl 231!ASSEMBLER_TYPE
usedInSource 231!ASSEMBLER_TYPE 0!ftfmctab.asm
FACT ATTRIBUTE:
0!ftfmctab.asm {
    dateChanged = 19970101
    path = /usr/bigsystem/version4/source/
    size = 329
    version = 1}
231!ftf_tbl {
    endChar = 238
    isConst = 0
    startChar = 231}
```

Figure 34: Description of an Assembly Language Program in TA++ Format

---

of an assembly language program is shown in TA++ format. The `FACT ATTRIBUTE` section demonstrates how TA++ (and also TA) supports loosely wrapped text for the list of attribute settings.

7 Towards a Standard Exchange Format

In Section 5.4 we noted that efforts towards integrating reverse engineering tools have been severely hindered by the lack of a consistent model for the structural makeup of software representations. The development of a Standard Exchange Format (SEF) for software is seen as the desired solution to this problem. Schemas that accommodate the representational needs of various reverse engineering tools are an essential part of an SEF. In this chapter we justify their necessity and classify them according to two distinguishing characteristics. Based on the use of schemas, four different exchange patterns are distinguishable. The relative benefits and drawbacks for each exchange pattern are discussed along with examples of their use within existing exchange formats. Next we evaluate how each of the exchange patterns satisfies thirteen requirements that an SEF would need to fulfill to make it attractive to tool developers. From this we determine the exchange pattern that is best suited for standardization into an SEF. Finally, we discuss a number of other challenges related to software representations that hinder efforts to integrate reverse engineering tools.

7.1 The Need For Schemas

Much of the discussion related to the establishment of an SEF in the literature and at conferences and meetings has centered not so much on how to negotiate exchange, but rather on what information model should be used [DE 94a, EKW 99, God 01, RW 91, Sim 00b]. More recently this discussion has focused on the makeup of a standardized model that would form the representational basis for an SEF [Sim 00c]. But can a single information model accommodate the needs of the entire reverse engineering community while at the same time satisfy all the requirements listed above? The answer is no for a number of reasons. The most significant deterrent relates to semantic differences that exist between models for different programming

languages [DDT 99]. Many programming languages have fundamental differences that cannot be accommodated in a single information model. For instance, in object-oriented languages such as Java all entities are organized within a hierarchy of classes. Instantiation outside the class hierarchy is not possible. Modular languages such as COBOL allow the creation of global external variables and records. These entities are nonexistent in object-oriented programming languages.

Some reverse engineering tools create extensions to software representations for storing proprietary information. These extensions may be solely for the internal use of the tool that created them or other selected tools may use them. For this reason there is a need to preserve the persistence of proprietary extensions in an exchange format, even if the destination tool does not make use of them. A single information model tends to force tool developers to subscribe to a rigid structure for representing software that does not permit the inclusion of proprietary extensions.

A single information model cannot capture all the views of software supported by reverse engineering tools [EKW 00]. In Section 6.1.2 we identified three levels of abstractive detail that are commonly used within the reverse engineering community. But there is a lot of debate over how many levels of abstraction are appropriate. Some reverse engineering tools look at software from unique perspectives that cannot be characterized within an echelon of abstraction. Certain types of metrics tools fall into this category. A single information model that accommodates all views of software in existence today would still be insufficient for future needs. This is especially important when considering the use of software within the context of different application domains. For example, the software that supports financial systems, user interface systems and scientific computing systems all have different characteristics.

What is needed is a representational foundation that is flexible so that it can accommodate support for a wide variety of representations for software. The use of E-R models constrained using schemas provides this flexibility. E-R models provide the generic means for representing software as a graph. The schemas apply constraints to the E-R model depending on the representational requirements of the application domain. Standardization comes through schema design rather than through adherence to a monolithic information model.

7.2 Schema Classification

We have established the need for schemas as a means for providing flexible support for the representation of software systems in various application domains. Ultimately the way a schema is used in an exchange format dictates how a tool will negotiate exchange among other tools that use the format. We classify the use of schemas into two categories: *schema definition* and *schema locality*.

7.2.1 Schema Definition

The Schema Definition category characterizes *how* the schema is defined. Within this category two classifications are identified:

Implicit. The structure of the representation is implied by the context in which the representation is used.

Explicit. The structure of the representation is provided, either through a specification or some other means.

7.2.2 Schema Locality

The Schema Locality category distinguishes *where* the schema is defined. Within this category two classifications are identified:

Internal. The schema is an integral part of the tool. As a consequence, the schema is not required to participate in the exchange.

External. The source for the schema definition is external to the tool. Because of this detachment, the schema is a participant in the exchange that occurs between tools. The schema is received by each of the tools either simultaneously with instance data or separately as a precursor to subsequent transmissions of instance data.

7.3 Exchange Patterns

According to the schema classifications outlined above, four different types of exchange can be negotiated among reverse engineering tools. We refer to these types of exchange as *exchange patterns*. In this section we characterize the exchange patterns and provide examples for each. The figures below provide a visual perspective on the exchange pattern as it relates to each individual tool participating in an exchange episode. Each figure consists of the following components:

- A tool `T`.
- A schema `S`.
- `I` and `I'` respectively representing the state of a software representation instance before and after it is processed by tool `T`.

In Figure 35 an exchange that uses an implicit/internal schema is shown. The schema is embedded in the code, so it is found in many locations within the tool. Tools that are built to make use of an API to exchange software representations fall into this category. APIs essentially have a fixed schema, so the tools that use them are constructed according to an implicit yet predetermined concept of the software representation being exchanged.

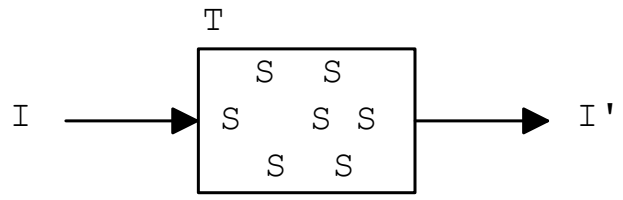


Figure 35: Exchange Using An Implicit/Internal Schema

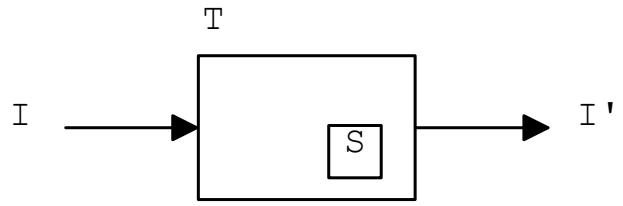
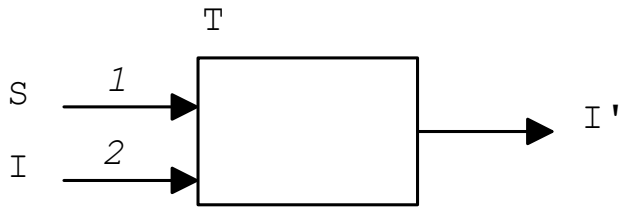
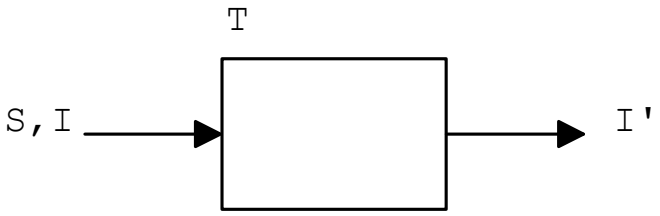


Figure 36: Exchange Using An Explicit/Internal Schema



(a) Consecutive receipt of schema and instance(s)



(b) Simultaneous receipt of schema and instance

Figure 37: Exchange Using An Explicit/External Schema

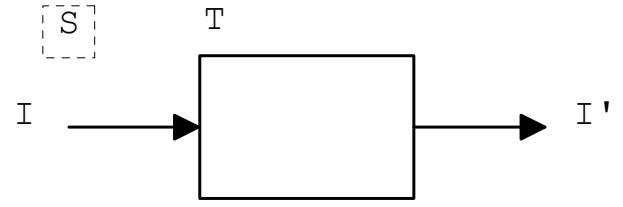


Figure 38: Exchange Using An Implicit/External Schema

Exchange that uses an explicit/internal schema is shown in Figure 36. Although the schema remains an integral part of the tool, it is provided as a specification so the schema is shown in a single location. Tools constructed within the PROGRES environment make use of schemas in this fashion. The tool developer first provides a schema in the form of a specification that outlines the graph-based structure of data to be represented and the operations that can be performed on them. Transactions that work with graph instances provide the functionality for the tool being constructed.

In Figure 37 we see two exchange varieties that use an explicit/external schema. The tool shown in (a) receives the schema first followed by the instance data. The exchange format GraX works in this fashion. In GraX and GXL, schema and instance data are stored separately. All data instances provide a link to the file where the schema is stored. In (b) the schema and the instance data are received simultaneously. TA works this way. The schema information stored in the scheme tuple and scheme attribute sections are exchanged with instance data stored in the fact tuple and fact attribute sections of the same file.

Exchange that uses an implicit/external schema is shown in Figure 38. In this case, the schema does not 'exist' (so it is shown in a box with a dashed border) yet it does dictate the structural semantics of the information exchanged. RSF is an example of an exchange format that works this way. The tuple notation used in each of the four RSF flavours is a syntactic requirement. The implicit schema for the information exchanged is an unconstrained E-R model. Tools such as *Rigi* and Holt's *Grok* [Hol 98d] accept E-R models in RSF. These tools have been pre-configured to handle constraint-free E-R instance data, so there is no need for a schema.

The exchange pattern used by each of the exchange formats we outlined in Chapter 6 are summarized in Table 4.

		Schema Definition	
		Implicit	*Explicit*
Schema Locality	*Internal*	ASIS CORUM CORUM II IML	ATerms PROGRES
	External	RSF	Datrix-TA[1] FAMIX[2] GraX[2] GXL[2] RG[1] TA[1] TA++[1]

Table 4: The Exchange Pattern Used By Each Exchange Format

7.4 Considerations

As we mentioned in Section 7.2, schema definition is a characterization of *how* the schema is defined, while schema locality relates to *where* the schema definition takes place. In this section we consider the advantages and disadvantages of each schema classification category in relation to their use in exchange.

7.4.1 Implicit Schema Definitions

The main advantage of an implicit schema definition is that it provides good performance. There is no need to carry out any additional processing or manage specifications for the representation being used. When the implicit definition is located internally, the representation is close at hand, being built into the code for the tool. Even when the implicit definition is external, the tool knows the structure of the information being exchanged so the opportunity to handle it appropriately is provided. This typically translates into the ability to processes large quantities of

[1] Employs simultaneous receipt of schema and instance.
[2] Employs consecutive receipt of schema and instance(s).

information in a fast and efficient manner. This is especially beneficial for reverse engineering tools that work with source whose magnitude is measured in millions of lines of code.

A number of disadvantages offset the performance advantages of an implicit schema definition. Because the definition is static by nature, the representation is not extensible. This is a major problem for tools that are built around a particular information model. In such a situation, making changes to the representation involves a wholesale revision of code. Documentation is also a problem when the schema definition is implicit. A separate document outlining the structure and semantics of the representation is a necessity. Maintaining this documentation is time consuming and keeping it in sync with tool or exchange format changes is especially challenging.

A third problem with implicit schema definitions relates to the manner in which tools typically accept input from the exchange channel. It is often useful to verify the integrity of information being exchanged. This usually involves a check to ensure that the input is well formed. When the schema definition is implicit, such a test is difficult to implement and maintain. The tool functions that handle representations are often deeply embedded and widely distributed throughout the code for the tool. A test that effectively checks incoming information must be based on all uses of the representation by the tool. In addition, the check must stay in sync with any modifications that are made to the exchange format or the internal representation within the tool over time. The effort involved in creating such a check in essence duplicates the efforts originally involved in handling the representation within the tool in the first place. As a consequence, it is unlikely that a tool that negotiates exchange using an implicit schema definition will include a check for well-formed input.

7.4.2 Explicit Schema Definitions

Exchange involving an explicit schema definition offers many benefits. The tool makes use of a specification or some other explicit means that identifies the structure and semantics of the information input from the exchange channel. A clear separation exists between schema and instance data, no matter if the schema definition is internal or external. Because the definition is dynamic by nature, the representation is highly extensible. Modification of the representation is easily accomplished through changes made to the schema specification. All the information relating to the representation is located in a single location. This makes it easier for humans to get an overall understanding of the structure and semantics supported. The representation is always well documented and up to date. The explicit definition for the schema is itself the documentation.

Checking for well formed input is a straightforward process when the schema definition is explicit. The schema specification holds all the requirements that must be satisfied for the information to pass the checker. Implementing the checker is simple because the schema specification is complete and close at hand. The checker does not need to be maintained because the schema specification always outlines the current representation in use. For these reasons, it is likely that a tool that negotiates exchange using an explicit schema definition will include a check for well-formed input.

The main drawback of an explicit schema definition is that it requires interpretation. The schema must be processed first before the tool can accept information from the exchange channel. This intermediate step ultimately affects the performance of the tool. More importantly, there is a requirement for the tool to ‘orient’ itself towards the representation provided in the schema. The tool must be flexible to accommodate this kind of functionality. In a best-case

scenario, the tool would be able to accommodate any representation. In reality, it is likely that the representational capabilities of many tools will be limited. Building flexibility into a tool may also add significant complexity to the development effort.

7.4.3 Internal Schemas

The main advantage of an internal schema is accessibility. The tool does not need to venture out to an external source to determine the structure and semantics of the information model. This is an obvious advantage in terms of performance.

The difficulty with an internal schema definition becomes apparent when there is a need to change the information model. Maintaining conformity among two or more tools is difficult to achieve. This is especially challenging when the schema definition is implicit in all the affected tools. Changes must be implemented exhaustively throughout the code for each of the tools affected. Clearly an internal schema tends to make all tools participating in the exchange conform to a rigid representational structure and semantics.

7.4.4 External Schemas

With an external schema definition, managing conformity among two or more tools participating in the exchange is easily accomplished. A single schema definition is all that is necessary to ensure that each tool is using the correct structure and semantics for the representation being exchanged. Complete representational conformity among each tool participating in the exchange is assured as long as each tool makes use of the same schema definition. An external schema definition eliminates the need for a complete overhaul of the code for each tool when a change is made to the representation.

Nevertheless, the rules that each tool uses to process and analyze exchanged information can come out of sync with the schema because its definition is separated from the tool. Maintaining

		Advantages	*Disadvantages*
Schema Definition	*Implicit*	• High performance no matter how large the input	• Not extensible • Hard to document • Difficult to implement check for well formed input
	Explicit	• Highly extensible • Easily understood • Well documented • Check for well formed input is easier to implement	• Low Performance • Tool code is more complicated
Schema Locality	*Internal*	• High performance	• Difficulties managing changes among two or more tools
	External	• Easier to manage changes among two or more tools	• Keeping the code consistent with the schema is difficult

Table 5: Advantages and Disadvantages of Schema Definition and Locality on Exchange

consistency between the code for a tool and the schema is challenging. The problem is exacerbated by the fact that external schemas are easily changed. The more often a schema is changed, the more likely that a loss of consistency will occur. The code in essence defines what the tool does with the information once it is successfully exchanged. But how this is accomplished is completely dependent on the structure and semantics of the representation defined externally by the schema.

7.4.5 Comparative Summary

Table 5 summarizes the relative advantages of disadvantages of each of the schema classification categories mentioned in this section.

7.5 Exchange Pattern Satisfaction of SEF Requirements

Requirements of an SEF that would best support integration and interoperability among reverse engineering tools have been widely considered (for example, see [TDD 00], [BJO 00], [Let 98] and [MWT 94]). In particular, St-Denis, Schauer and Keller [SSK 00] list 13 requirements for an exchange format based on their past experiences and various requirements outlined in [BGH 99,

1. **Transparency**. No loss, alteration or gain in the information transferred occurs due to the use of encoders and decoders by the exchange format.

2. **Scalability**. The format is usable for exchanging information of all sizes, including representations for very large software applications.

3. **Simplicity**. The format is not complex or intricate. This makes it efficient, easier to describe, comprehend, apply and maintain while statistically reducing the prospects for errors and making it easier to process in an automated fashion.

4. **Neutrality**. The representation is independent so that as many tools as possible can integrate with it.

5. **Formality**. A formal definition reduces the chances for misinterpretation and ensures that it is well understood by all parties.

6. **Flexibility**. The format accommodates different tools, languages and syntax for data and schemas. It also accommodates the exchange of incomplete information.

7. **Evolvability**. The format can be changed easily to accommodate future needs.

8. **Popularity**. Adoption of the format is widespread so that as many tools as possible can take advantage of it.

9. **Completeness**. Everything needed to exchange information successfully is included. The user does not have to look after details relating to the exchange.

10. **Schema Identity**. Transformation of instance data while preserving its identity is supported. The exchange format is capable of converting instance data from one schema into instance data of another schema. The instance data remains the same; it is just represented differently from one schema to the next.

11. **Solution Reuse**. Wherever possible, use existing techniques and methods with the goal of reducing the amount of time and effort spent in testing and deploying the format.

12. **Legibility**. A human reader can easily understand the format.

13. **Integrity**. Special mechanisms ensure information is exchanged without errors.

Source: [SSK 00]

Table 6: Requirements For A Standard Exchange Format

×	The exchange pattern does not satisfy the requirement.
✓	The exchange pattern satisfies the requirement.
✓✓	The exchange pattern satisfies the requirement in a way that is particularly beneficial.
–	The exchange pattern neither satisfies nor does not satisfy the requirement because it does not relate to the requirement.

Table 7: Markers Used to Indicate SEF Requirement Satisfaction

Bra 98, Cov 98, Hol 98a, LA 97, Mül 98, OMG 98, WCK 99]. These are shown in Table 6. Based on these requirements, in this section we identify the exchange pattern that is best suited for standardization in an SEF.

Each requirement from [SSK 00] is listed in a separate sub-section below. A short description followed by an evaluative statement of the merits or shortcomings of each exchange pattern is provided. Visual markers (shown in Table 7) are used to provide an overall indicator how the exchange pattern satisfies the requirement.

7.5.1 Transparency

No loss, alteration or gain in the information transferred occurs due to the use of encoders and decoders by the exchange format.

Implicit/Internal (–)

Explicit/Internal (–)

Implicit/External (–)

Explicit/External (–)

This requirement specifically deals with information handling procedures at both ends of the exchange channel. None of the exchange patterns involve the use of encoders or decoders.

7.5.2 Scalability

The format is usable for exchanging information of all sizes, including representations for very large software applications.

Implicit/Internal (✗)

Although an implicit schema definition provides high performance, the fact that the schema definition is embedded in the code means that the tool's capacity is fixed. Making variations to the code to accommodate different magnitudes of information is difficult.

Explicit/Internal (✓)

Although the explicit schema definition reduces the performance of the tool, it provides flexibility that makes it easier to adjust the representation to address scalability issues. For instance, one strategy for managing large bodies of information is to exchange only specific pieces of it rather than the whole thing. When the schema definition is explicit, adjusting the amount of information exchanged is much easier than when the definition is implicit.

Implicit/External (✗)

The implicit schema definition provides high performance but once again the tool is tailored to handle information in a particular way only. Although the schema might be easy to change because it is external, the tool may not be able to handle large volumes of information without significant code changes.

Explicit/External (✓)

The advantages of this exchange pattern are identical to those for the explicit/internal exchange pattern, although the performance degradation may be more significant because the locality of the schema is external.

7.5.3 Simplicity

The format is not complex or intricate. This makes it efficient, easier to describe, comprehend, apply and maintain while statistically reducing the prospects for errors and making it easier to process in an automated fashion.

Implicit/Internal (✗)

The schema is indeed complex and intricate, being disseminated throughout the code for the tool. It is difficult to understand and maintain making it prone to erroneous modification.

Explicit/Internal (✓)

The non-embedded nature of the schema specification simplifies the exchange and makes it much easier to understand and maintain. The close proximity of the schema to the tool code provides greater efficiency over the explicit/external exchange pattern.

Implicit/External (✓✓)

The schema does not 'exist' which simplifies the exchange process and provides an environment where the throughput of information can be maximized.

Explicit/External (✓)

The schema specification simplifies the exchange process, but its separation from the tool makes it less efficient than the explicit/internal exchange pattern.

7.5.4 Neutrality

The representation is independent so that as many tools as possible can integrate with it.

Implicit/Internal (✗)

There is no neutrality of the representation as it is embedded into the code for the tool.

Explicit/Internal (✗)

The explicit nature of the schema definition provides a degree of neutrality. Nevertheless, the schema locality is internal to the tool, which impedes the integration of other tools to a standard representation.

Implicit/External (✗)

The schema is independent from the tool, which provides it with some degree of neutrality. Nevertheless, the schema is non-existent, so it is difficult to define a standard for other tools to integrate with it.

Explicit/External (✓✓)

Neutrality is maximized. The schema definition is completely separate from all tools and is explicitly defined. This makes it easier to integrate other tools to a standard representation.

7.5.5 Formality

A formal definition reduces the chances for misinterpretation and ensures that it is well understood by all parties.

Implicit/Internal (✗)

There is no formal definition of the representation so it is very difficult to transfer knowledge of it to others. This is especially problematic because of the embedded nature of the representation.

Explicit/Internal (✓)

By default, the explicit schema definition is formal. Because it is internal to the tool, all concerns relating to the tool implementation are together in the same place.

Implicit/External (✗)

The implicit nature of the representation means there is no formal definition. Because the schema locality is external, documentation must be relied upon for information on the representation.

Explicit/External (✓✓)

The explicit schema definition is itself a formal means for expressing the structure of the data instances. The external schema locality makes it easier for all tool integrators to understand the representation.

7.5.6 Flexibility

The format accommodates different tools, languages and syntax for data and schemas. It also accommodates the exchange of incomplete information.

Implicit/Internal (✗)

The exchange pattern offers no flexibility at all. The tool itself must handle any accommodation for different tools, languages or data/schema syntax.

Explicit/Internal (✓)

The explicit schema provides flexibility for changing the representation. This is partially negated by the fact that the schema is defined internally, which ties it very closely to the tool it is contained in.

Implicit/External (✗)

A certain degree of flexibility is offered by the implicit schema definition because it is external from all tools that participate in the exchange. Nevertheless, because the schema definition is implicit, it is difficult to offer representational flexibility. Each tool must conform to the same non-existent schema definition. This tends to force developers to keep to a rigid representational standard.

Explicit/External (✓✓)

Flexibility is maximized. First the schema definition is external, so it is not tied to any one tool. Second, the schema is explicitly defined so the representation is clear and easily modified.

7.5.7 Evolvability

The format can be changed easily to accommodate future needs.

Implicit/Internal (✗)

Change is difficult to manage because the representation is embedded in the code for each tool.

Explicit/Internal (✗)

Although the explicit schema definition supports evolutionary changes, the internal locality of the schema ties the representation too closely with the tool. Changes to the representation must be implemented on a tool-by-tool basis.

Implicit/External (✗)

Although the schema definition is located externally, change is difficult to accommodate because the schema definition is implicit. Evolutionary changes are difficult to implement when all parties involved must approve it.

Explicit/External (✓✓)

Evolvability is maximized. The external schema definition does not tie the representation to any one tool. The explicit definition encourages evolutionary change in a collaborative manner.

7.5.8 Popularity

Adoption of the format is widespread so that as many tools as possible can take advantage of it.

Implicit/Internal (–)

Explicit/Internal (–)

Implicit/External (–)

Explicit/External (–)

Exchange patterns that have external schemas may become more popular because they facilitate the use of well-accepted document exchange methods such as XML. Nevertheless, the success of a particular exchange format ultimately rests with those who use it within the reverse engineering community.

7.5.9 Completeness

Everything needed to exchange information successfully is included. The user does not have to look after details relating to the exchange.

Implicit/Internal (–)

Explicit/Internal (–)

Implicit/External (–)

Explicit/External (–)

We have differentiated between schema and instance data in the exchange process. Although these two components are required to carry out exchange (and in this way they *typify* how the exchange is managed) they do not represent a complete exchange format.

7.5.10 Schema Identity

Transformation of instance data while preserving its identity is supported. The exchange format is capable of converting instance data from one schema into instance data of another schema. The instance data remains the same; it is just represented differently from one schema to the next.

Implicit/Internal (✗)

The implicit nature of the schema definition makes it very difficult to support transformation of instance data. The use of the representation is embedded into the code for the tool. Identifying instance data and transforming it into an equivalent alternate representation is challenging.

Explicit/Internal (✓✓)

The schema definition is explicit which greatly assists in identifying the structure and semantics of instance data. At the same time, the schema is internal so it reflects the tool's view of instance data. Transformation of this schema to an external schema for exchange is all that is necessary.

Implicit/External (✗)

Once again, the implicit nature of the schema definition makes it very difficult to support transformation of instance data. The external schema definition is non-existent, which makes it difficult to identify a transformation to another schema that will preserve the identity of the instance data.

Explicit/External (✓✓)

The explicit schema definition lays out the representation in a single location external to the tool. Schema transformation can be carried out away from each of the tools participating in the exchange.

7.5.11 Solution reuse

Wherever possible, use existing techniques and methods with the goal of reducing the amount of time and effort spent in testing and deploying the format.

Implicit/Internal (✗)

The representation is embedded into the tool code, which makes it very difficult to reuse.

Explicit/Internal (✗)

Although explicitly defined, the representation remains closely tied to the tool. This tool centricity makes it difficult to reuse the representation outside the tool environment.

Implicit/External (✗)

The non-existent schema definition is not easily described which makes it difficult to reuse.

Explicit/External (✓)

The representation is defined explicitly and is not tied to any one tool. This tends to make it easier to reuse and makes it easier to implement and test.

7.5.12 Legibility

A human reader can easily understand the format.

Implicit/Internal (✗)

The embedded nature of the representation makes it difficult to understand, especially for non-programmers. Well-documented code may partially offset this problem.

Explicit/Internal (✓)

Understanding of the representation is much easier when it is explicitly specified in a single location within the tool.

Implicit/External (✗)

The non-existent nature of the schema definition impedes understanding of the representation. This combined with the fact that the schema locality is external means that independent documentation must be relied upon to get information about the representation.

Explicit/External (✓)

An explicit schema definition eases the legibility of the representation. The external locality of the schema ensures that the representation is tool independent.

7.5.13 Integrity

Special mechanisms ensure information is exchanged without errors.

Implicit/Internal (–)

Explicit/Internal (–)

Implicit/External (–)

Explicit/External (–)

The integrity of the exchange ultimately rests on the underlying technology used to communicate information.

7.5.14 Comparative Summary

Table 8 summarizes our evaluation of the SEF requirements listed in [SSK 00]. It is clear that the use of an explicit schema definition with external schema locality satisfies all the requirements that relate to exchange. In fact, five of the nine exchange-related requirements are strongly satisfied by the explicit/external exchange pattern. Following a distant second is the

		Exchange Pattern			
		Implicit/ Internal	*Explicit/ Internal*	*Implicit/ External*	*Explicit/ External*
Requirements	*Transparency*	–	–	–	–
	Scalability	×	✓	×	✓
	Simplicity	×	✓	✓✓	✓
	Neutrality	×	×	×	✓✓
	Formality	×	✓	×	✓✓
	Flexibility	×	✓	×	✓✓
	Evolvability	×	×	×	✓✓
	Popularity	–	–	–	–
	Completeness	–	–	–	–
	Schema Identity	×	✓✓	×	✓✓
	Solution Reuse	×	×	×	✓
	Legibility	×	✓	×	✓
	Integrity	–	–	–	–

Table 8: Exchange Pattern Satisfaction of SEF Requirements

explicit/internal exchange pattern. Both of the exchange patterns with an implicit schema definition are the least satisfactory. Between these two, the implicit/external pattern is strongly beneficial solely because of its simplicity.

To summarize our evaluation, the use of schemas with an explicit definition and external locality are the preferred choice for an SEF. The schema definition appears to be the most important factor in the evaluation. Schema locality matters as well, but it is a more minor option to consider.

7.6 Additional Challenges

So far we have shown that schemas are an essential component in the exchange process, their use defining how tools interact with each other. After a critical evaluation of schema use, we have concluded that schema definitions that are explicit and external best facilitate exchange among reverse engineering tools. Nevertheless, other problems exist that relate to the software representations themselves that hinder exchange and integration efforts.

7.6.1 The Naming Problem

E-R models require that unique names known as identifiers be assigned to each entity instance. When integrating information from two or more tools it is important that all identifiers be reconciled. Two problematic situations arise [BGH 99, Let 98]

1. **Aliasing**. More than one identifier is used for the same entity instance.
2. **Nondiscrimination**. The same identifier is used to represent different entities. This is most problematic when entities with the same name occur among different scopes within the program.

Together these two situations exemplify what is known as the *naming problem* [BGH 99]. One way to get around the naming problem is to create an identifier based on the name of the entity the identifier refers to. But this solution breaks down easily, since some programming languages allow the same name to be used for different program elements. Another solution known as *mangled names* creates an identifier based on a combination of the class name for the entity and its corresponding scope information. Compilers and linkers use special algorithms for generating mangled names. But they differ from one programming language to the next and they do not usually support unique naming for local variables and data types.

Clearly, if tool integration is ever to be successful, a consistent scheme for uniquely identifying entities within a software representation must be established [God 99, God 01].

7.6.2 Target Resolution

In E-R models, references among entities in a software representation are recorded as relations. *Target resolution* refers to the identification of the entity that a given entity relates to. Tools operate differently in the way they carry out target resolution. In general one of the following approaches are used [BGH 99, God 99]:

Declaration. The reference targets the declaration in a header file.

Static Definition. The reference targets the static definition in the body of the code.

Dynamic Definition. The reference targets the dynamic definition (as is the case for virtual functions and pointers).

No target resolution. No record of contextual information about an entity is made. This makes it impossible to resolve relation references. Each reference to a source code element results in the creation of a new entity.

Inconsistencies in the way that tools perform target resolution can hinder tool integration efforts. For example, consider the challenge involved in integrating instance data from a tool that performs no target resolution with instance data from a tool that performs declaration target resolution. Depending on the number of references in the source code, the former will have a number of redundant entities. This redundancy must be eliminated before the information can be merged with information from the second tool.

Methods for integrating information from tools that perform different types of target resolution do not always work correctly. This can lead to a situation where the entity referred to in a relation is the wrong one.

7.6.3 The Line Number Problem

It is often desirable to relate entities in a representation back to the place in the source code where they originated. The physical line number within the source code is commonly used as the location identifier. But this is not the best approach because line numbering may be inconsistent among different systems. Worse still, a group of possibly non-contiguous lines within the source code may relate to an entity. For example, consider high-level architectural entities that originate from various clusters or conglomerations of source code.

These challenges relate to the *line number problem* [BGH 99, God 99, God 01]. To be able to integrate information from various reverse engineering tools, a common basis for referencing entities to source code elements must be established.

7.6.4 System-Wide Representation

Obtaining a system-wide representation for software involves managing various integration issues. For example, *Datrix* tools parse Java at the source code level, while C and C++ are parsed before they are fully preprocessed and linked. This means that analysis is carried out on each constituent artifact of the system, rather than on the whole thing all at once [LLL 01]. Obtaining a representation such an ASG for an entire system is non-trivial, involving at least two steps. First, the linkage relationships between libraries, executables and the system must be determined. Next the semantic relationships among the code segments must be determined. These relationships are not apparent until linking has occurred. Reverse engineering tools must support integration of information from all these sources.

Another problem relates to the analysis of modules of one or more systems. It is often the case that semantic links exist between two or more modules that affect the information that independent module analysis provides. An example is provided in Figure 39. Two design recovery instances from the source code for two different modules of a single system are shown. The function application `P(y)` has been recovered from module A, yet no information about `y` is provided. In module B a declaration for `x` and an application of `P` to `x` has been recovered. We can deduce that `y` from module A contains a date because `x` in module B contains a date and is a parameter for function `P`. The semantic link between each of the design recovery instances is `P(a)` representing the application of function `P` to the unknown parameter `a`. It is clear that a

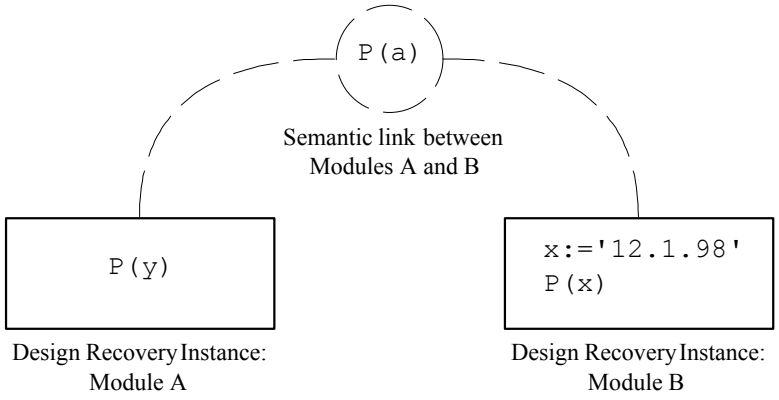


Figure 39: A Semantic Link Between Submodules A and B

---

complete design recovery of module A is impossible unless it includes specific information integrated from the design recovery of module B.

8 The Road Ahead

So far we have provided an in-depth overview of existing formats for exchanging software representations among reverse engineering tools. The need for an SEF is widely acknowledged as a necessity for enabling data integration and tool interoperability. We have evaluated four exchange patterns in their ability satisfy the requirements for an SEF. In this chapter we summarize the steps that have been taken within the reverse engineering community towards the creation of an SEF and provide a perspective on the future direction these efforts will take. At the same time, we offer an alternative solution to the integration problem that may be the focus of our future research.

8.1 GXL – The Standard Representational Exchange Mechanism

In Section 6.2.8 we discussed GXL as an exchange format among others in use currently within the reverse engineering community. Like some of the other exchange formats, GXL provides a flexible means for exchanging all sorts of software representations. As we determined in Chapter 7, its use of explicit, externally defined schemas provides a number of advantages for integrating reverse engineering tools. Nevertheless, GXL is distinctive in a way that makes it particularly superior over other exchange formats.

GXL's E-R representational foundation consists of a three-layer hierarchy. This is shown in Figure 40. At the lowest level, data instances consisting of entities and relations are represented as E-R graphs. These graph instances are constrained by schemas defined at the schema level. Schemas are often represented as conceptual models, which are also E-R graphs. For example, the conceptual model for GXL attributes is shown in Figure 41. In this example from [Win 01], a UML class diagram is used to establish attribute entities and the relationships among them. The conceptual model is itself an E-R graph consisting of attributes that define multiplicities and

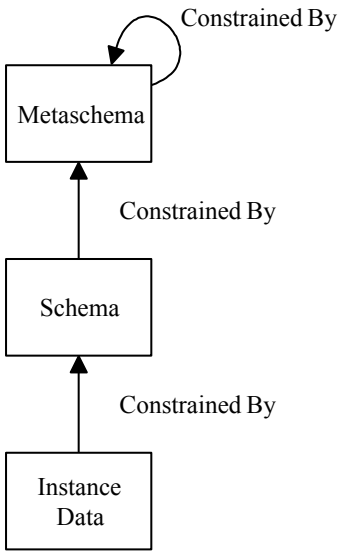


Figure 40: GXL's Three-Layer Representational Hierarchy

---

other information. These schema graphs are constrained by *metaschemas* defined at the metaschema level. Metaschemas are simply schemas that set out the characteristics for a class of schemas. In GXL, the metaschema is interesting because it is self-referential: an E-R graph is used to define the allowable characteristics for GXL schemas, which are also represented as E-R graphs.

The metaschema layer is the main advantage that GXL has over other formats for exchanging software representations. The metaschema definition is a consistent foundation from which all other schema level representations can be defined. So in the case of GXL, a metaschema for E-R graphs provides a common base from which any schema for representing software is derived. A consistent means for defining and exchanging software representations is built into GXL. For this reason the reverse engineering community has embraced GXL as a standard representational exchange mechanism. Although the E-R metaschema for GXL is not yet complete, enough of it is in place so that work on creating standard schemas for representing software can begin.

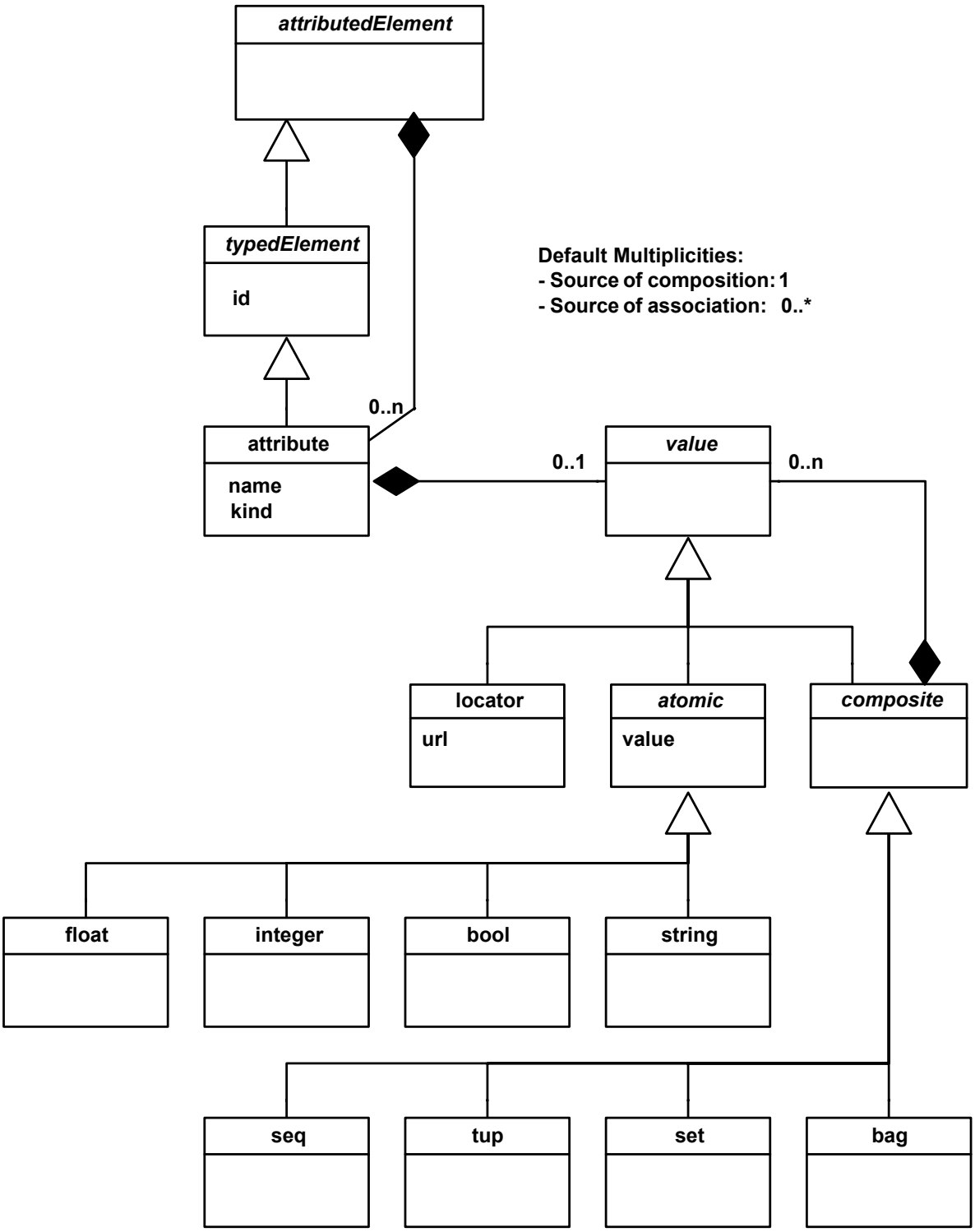


Figure 41: A Conceptual Model for GXL Attributes

8.2 The Next Step – Standard Schemas for Software Representations

Now that GXL has been established as the format of choice for exchanging software representations, the next step involves the creation of GXL schemas for the representations themselves. Much of the work in this area has focused on defining representations at the various levels of abstraction. As we discussed in Section 6.1.2, three levels of abstraction are referred to in general within the reverse engineering community: *low*, *medium* and *high*.

At the lowest level of abstraction, the ASG representation established by the Datrix group is popular. This is evidenced by tools such as CPPX that translate the IR from the Gnu GCC [Gnu 01] compiler into Datrix [CPP 01, DMH 01]. Nevertheless, discussion on a standard schema for representing programming languages such as C and C++ continues [FGS+ 01]. In particular, the schema used for representing software in the Columbus system is an ongoing effort [Fer 01, FMB+ 01].

At the medium level of abstraction, the *Dagstuhl Middle Model (DMM)* [Let 01] has been developed in an ongoing fashion since the *Dagstuhl Seminar on Interoperability of Reengineering Tools* took place in January 2001. DMM consists of four classes of hierarchical representations:

1. **Top Level Model**. Provides an overall view of the DMM hierarchy of classes. The current DMM top-level model shown in UML from [Let 01] is shown in Figure 42.
2. **ModelObjects**. Provides an abstract view of a program ignoring the layout of the source code.
3. **SourceObjects**. Refers to sections of source code that define, declare and refer to ModelObjects.
4. **Relationships**. Defines relationships among SourceObjects and ModelObjects.

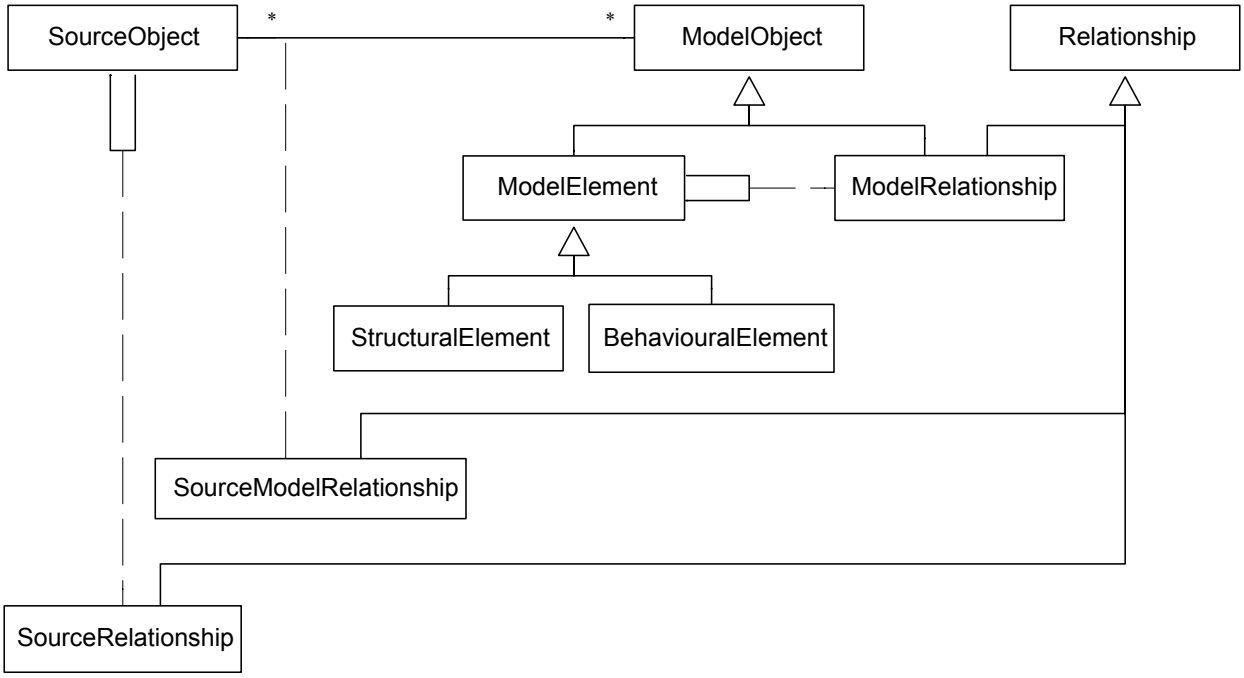


Figure 42: The DMM Top-Level Model

DMM is not yet complete. Aside from the representational makeup of the model, a significant issue that needs to be resolved relates to what a minimal representation consists of. When a DMM instance is exchanged, which elements are required and which are optional? This is in fact an important issue for all representations that are exchanged. Reverse engineering tools differ in the information that they process. When these tools are integrated together, some will not recognize certain parts of an exchanged representation. Other parts of the representation that a tool might expect to be present may not be. Tools that participate in an exchange must be capable of operating with insufficient or incomplete data while at the same time be ready to look after extraneous information that might exist in a representation.

Standard GXL schemas for representing software at high levels of abstraction have not yet been explored. Nevertheless, RSF or TA may provide appropriate high-level schemas as some design recovery tools for representing architectural concepts currently use them. For example,

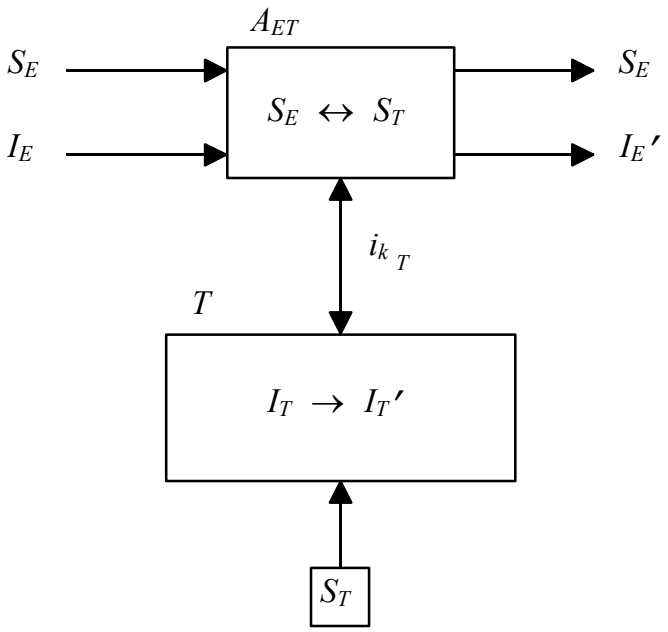
the *TA Exchange Format (TAXForm)* is a repository-based system for sharing information extracted from source code between reverse engineering tools that work at the architectural level [BGH 99, God 99, God 00b]. It makes use of TA for the language syntax, making use of schemas for structuring the representation of information in the repository.

8.3 The Need for Adaptive Integration

Admittedly, our discussion on schema use and exchange patterns has so far focused on a conformist approach to integration. Success of an SEF relies heavily on the willingness of tool developers to agree on a common schema for their representational requirements. So even though GXL has been accepted as a standard exchange mechanism, there is still much work to be done in coming up with schemas for all the different representations that integration among reverse engineering tools would require.

A better approach is to take an adaptive view of integration. Each tool uses their own internal schema that defines the structure, semantics and constraints for the information that they work with. An adapter maps the internal tool schema to an external schema. This external schema defines the structure, semantics and constraints for the information that is exchanged. Once a mapping adapter from the internal schema to the external schema is defined, the data is easily transformed to follow suit.

Figure 43 shows how adaptive integration could be used to make a software representation carried by an exchange format available to a reverse engineering tool. Consider an exchange format E that consists of a schema (S_E) and instance data (I_E). I_E consists of many individual data instances (i.e. factual information about a software system) that conform to S_E. Each of these instances are indicated as i_{n_E}. So the expression $\{i_{1_E}, i_{2_E}, \ldots, i_{n_E}\}$ indicates the collection of individual data instances that together make up I_E.



$$I_E = \{i_{1_E}, i_{2_E}, \ldots, i_{n_E}\}$$

$$I_T = \{i_{1_T}, i_{2_T}, \ldots, i_{n_T}\}$$

Figure 43: An Adaptive Integration Example

Tool T processes data instances that are structured according to it's own internal schema S_T. The problem is that T cannot access I_E because of differences between S_E and S_T. An adaptive integration solution involves the creation of adapter A_{ET}. This adapter provides a means for transforming I_E instances to I_T instances and vice-versa on an individual basis. On request, adapter A_{ET} provides tool T with individual instances from I_E in a form that is compliant with S_T. The tool processes the information and then returns the changes to A_{ET}. The adapter then transforms the individual I_T instances into a form that is compliant with S_E. Within this scenario, the real challenge comes in integrating the changed instance data with the original instance data. Aside from managing the replacement of modified information, the adapter must also look after updating all dependencies that are impacted by the changes made.

The main advantage of the adaptive approach to integration is that it does not force any change to the internal representation used by an existing reverse engineering tool. All the integrative functionality is built into the schema adapter. A major developmental effort comes in the creation of the adapter itself. But once this is accomplished, mapping data from one schematic convention to another would be a straightforward process.

9 Conclusion

The purpose of this paper was to examine the exchange of software representations among tools that support reverse engineering. To start off, we took an in-depth look at maintenance to get a grasp on how it is carried out and to establish the domain within the field of software where reverse engineering is practically applied. We examined how tools support reverse engineering activities and how they are constructed to gain insight into how software is represented. The need for integration and interoperability among reverse engineering tools was outlined and three approaches to integration were presented. The most important aspect of this discussion centered on the need for a standard means for representing software. Without an agreeable set of criterion for representing software, concise, efficient exchange of information among reverse engineering tools is not possible. We reviewed the characteristic properties of fourteen formats currently is use for exchanging software representations among reverse engineering tools. The need for schemas and their use in software exchange formats were discussed. In classifying schema use, we identified four patterns of exchange. These were evaluated on their own merits and against a set of fourteen requirements for a standard format for exchanging software representations. Some other representational issues that impede integration efforts were also discussed.

We concluded by explaining the reasoning behind the selection of GXL as the standard representational mechanism for exchanging software representations. Although significant, this was not a monumental step in the progression towards a representational paradigm for software. Much work remains in coming up with standard schemas for representing software at various levels of abstraction. Our conclusive argument is that other means for enabling integration among reverse engineering tools may be feasible. To this end, adaptive integration may provide and alternative perspective on reverse engineering tool integration.

10 Glossary

There is often a great deal of variability in the use of terminology within the software engineering community. Fortunately, within the reverse engineering domain the use of terms outlined by Chikofsky and Cross [CC 90] have been widely accepted. For consistency we have adopted their terminology wherever possible. Definitions quoted directly from [CC 90] and other sources are appropriately indicated.

AST

Abstract Syntax Tree. A tree structure consisting of a hierarchy of nodes representing source code entities. Entities at the top of the hierarchy are decomposed into various sub-components via edges that identify the relationship between each of the nodes. The AST for a simple source code example is shown in Figure 4 on page 21.

ASG

Abstract Semantic Graph. A graph structure consisting of an AST that has been modified to eliminate indirect references to names and classes. Multiple occurrences of nodes with the same type are also combined into a single type node. The ASG for a simple source code example is shown in Figure 5 on page 21.

Abstract Syntax

The data structure used to represent software in an exchange format.

Abstraction

See *Levels of Abstraction.*

API

Application Programming Interface. An approach to integrating reverse engineering tools where one program provides the means for another program to interface to it.

ASIS

Ada Semantic Interface Specification. An open source API written in Ada95 for accessing the IR of the Ada95 Compilation Environment.

ATerms

Annotated Terms. An exchange format for representing data produced by parsers, structural editors, compilers and other components in software reengineering tools. A library of API functions for manipulating ATerms is also available.

CORUM/CORUM II

Common Object-based Re-engineering Unified Model. An API-based environment for integrating software reengineering tools. CORUM works at the source code level. CORUM II is a proposal for enhancing CORUM to provide advanced functionality for analysis of a system at the architectural level of abstraction

Datrix-TA

A format very similar to TA for exchanging software representations among the different tools that make up the Datrix system. See also *TA*.

DIF

Direct Inter-Tool Functionality. A transfer mechanism used by a software exchange format. Exchange between tools is facilitated through direct tool-to-tool interfaces.

DTD

An XML *Document Type Definition*. "A DTD is a formal description in XML Declaration Syntax of a particular type of document. It sets out what names are to be used for the different types of element, where they may occur, and how they all fit together." [Fly 01]. See also *XML*.

E-R Model

Entity-Relationship Model. Chen's [Che 76] convention for modeling relationships using graphs. Graph nodes represent entities that are relevant to a particular problem domain. Graph edges establish relationships among the entities that have been defined. Other properties that describe entities and relations are respectively added to the graph as node and edge attributes.

Exchange Format

An approach to integrating reverse engineering tools where an agreement is made between tool developers on the structure and meaning of information to be exchanged. A schema is typically used to define the information that the exchange format can consist of and how it should be interpreted [KCE 00b].

Exchange Pattern

Characterizes how exchange among reverse engineering tools is carried out. Each pattern is differentiated by how it makes use of schemas (expressed in terms of *schema definition* and *schema locality*). Four exchange patterns have been identified: *Implicit/Internal*, *Explicit/Internal*, *Explicit/External* and *Implicit/External*. Refer to Section 7.3 and in particular the examples on page 76 for more information.

Explicit/External Exchange

See *Exchange Pattern*.

Explicit/Internal Exchange

See *Exchange Pattern*.

Explicit Schema Definition

See *Schema Definition*.

External Schema Locality

See *Schema Locality*.

FAMIX

FAMOOS Information Exchange Model. A model developed at the University of Berne for exchanging object-oriented source code representations among reverse engineering tools.

Forward Engineering

"the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [CC 90]

GraX

An XML-based format developed at the University of Koblenz-Landau for exchanging software representations among their *KOGGE* and *GUPRO* tools.

GXL

Graph Exchange Language. A popular XML-based format for exchanging graphs (in particular graphs representing software) among a variety of tools. GXL is the product of a broad, collaborative effort within the reverse engineering community to come up with a

standard mechanism for exchanging software representations among reverse engineering tools.

HTML

HyperText Markup Language. A popular publishing language for formatting documents available on the World Wide Web.

IML

InterMediate Language. A portable IR developed by the Bauhaus Group that makes use of an attributed tree structure to represent details at the source code level.

Implicit/Internal Exchange

See *Exchange Pattern*.

Implicit/External Exchange

See *Exchange Pattern*.

Implicit Schema Definition

See *Schema Definition*.

Instance

Data represented as an E-R graph constructed in accordance to the conventions outlined in the schema for the class of graphs it belongs to.

Internal Schema Locality

See *Schema Locality*.

IR

Internal Representation. A representation for software stored internally in a compiler.

Legacy Information System

"Any information system that significantly resists modification and evolution." [BS 95]

Levels of Abstraction

A hierarchy of perspectives from which details about a software system can be viewed. Three levels of abstraction are commonly distinguished: *low*, *medium* and *high*.

Maintenance

The modification of a software system after it has been delivered.

Metaschema

A definition for a class of schemas. It lays out the characteristics that each schema within the class must conform to.

Migration

A maintenance activity that involves shifting a software system to a more adaptable and maintainable environment. The data and functionality of the original system is preserved. [BLW+ 99]

Portable IR

Portable Internal Representation. An approach to integrating reverse engineering tools that involves storing and exchanging software representations produced from the information stored internally in a compiler.

PROGRES

PROgramming with Graph Rewriting Systems. An environment consisting of an integrated set of freeware tools that help developers create, analyze, compile and debug specifications for graph rewriting systems.

Reclamation

See *Reengineering*.

Redevelopment

A maintenance activity that involves the replacement of an existing software system with a new one. [BLW+ 99]

Reengineering

A maintenance activity that involves "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [CC 90].

Renovation

See *Reengineering*.

Repository

A database where facts and detailed information relevant to the reverse engineering of a software system are stored.

Restructuring

A maintenance activity that involves "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)" [CC 90]

Reverse Engineering

"the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction." [CC 90]

RG

Resource Graph. An exchange format developed by the Bauhaus Group for representing source code at medium and high levels of abstraction.

RSF

Rigi Standard Format. A format for exchanging information among the components that make up the Rigi tool developed at the University of Victoria.

Schema

A definition for a class of graphs (typically E-R graphs) that lays out the allowable characteristics that each graph can take on. Graph constraints and node and edges semantics are typically outlined.

Schema Definition

A classification category for schema use that characterizes how a schema is defined. When the schema definition is *implicit*, the structure of a software representation is implied by the context in which the representation is used. When the schema definition is *explicit*, the structure of a software representation is provided, either through a specification or some other means.

Schema Locality

A classification category for schema use that characterizes where a schema is defined. When the schema definition is *internal*, the schema is an integral part of the tool so the schema does not participate in the exchange of information among tools. When the schema definition is *external*, the schema is detached from the tool and as a consequence must participate in the exchange.

Schema Type

Distinguishes software exchange formats between those that have a flexible schema (which can be altered to accommodate changes) and those that have a fixed schema.

SEF

Standard Exchange Format. A standard way of representing software for exchange among reverse engineering tools.

STS

Structured Text Stream. A transfer mechanism used by a software exchange format. Text stored in a structured format is packaged and exchanged from one tool to another through a communications medium such as the Internet

TA

Tuple-Attribute. A flexible language that makes use of a tuple notation for expressing graphs of large software systems.

TA++

A modified version of TA (see above) used for representing and manipulating software representations among components that make up the TkSee reverse engineering tool developed at the University of Ottawa.

Transfer Mechanism

The method used by a software exchange format to carry out data exchange. Three types of transfer mechanisms are discussed in this paper: *Files*, *Structured Text Stream (STS)* and *Direct Inter-Tool Functionality (DIF)*.

Type of Encoding

The data characteristics of the information exchanged by a software exchange format. Typically information is exchanged as text or in a binary format.

Wrapping

A maintenance activity that involves the encapsulation of an existing system with an outer shell, which acts as an interface to the system it encloses. [BLW+ 99]

XML

Extensible Markup Language. "XML is a markup specification language with which you can design ways of describing information (text or data), usually for storage, transmission, or processing by a program" [Fly 01]

11 References

[ADH+ 99] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy and C. Sapuntzakis. *The Basic SUIF Programming Guide*. Computer Systems Laboratory, Stanford University, November 1999.

[AE 95] M. Andries and G. Engels. *A Hybrid Query Language for an Extended Entity-Relationship Model*. Technical Report #TR95-03, Leiden Institute of Advanced Computer Science, Leiden University, Leiden, NL, January 1995.

[ASI 98a] Introduction to ASIS. 17-Aug-1998.
URL: `http://www.acm.org/sigada/wg/asiswg/intro.html`

[ASI 98b] ASIS Basic Concepts. 17-Aug-1998.
URL: `http://www.acm.org/sigada/wg/asiswg/basics.html`

[ASI 98c] ASIS Architecture. 17-Aug-1998.
URL: `http://www.acm.org/sigada/wg/asiswg/arch.html`

[Bau 01] The Bauhaus Project Home Page.
URL: `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/index-english.html`

[BFS 88] D. Baker, D. Fisher and J. Shultis. *The Gardens of Iris*. Technical report, Incremental Systems Corporation, Pittsburgh, PA, 1988.

[BGH 99] I. Bowman, R. Holt and M. Godfrey, Connecting architecture reconstruction frameworks. *Proceedings of the 1st International Symposium on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, May 1999, pp. 43-54.

[BJK+ 00] M. van den Brand, H. de Jong, P. Klint and P. Olivier. Effcient Annotated Terms. Software, Practice & Experience, 30:259-291, 2000.

[BJO 00] M. van den Brand, H. de Jong and P. Oliver. A Common Exchange Format for Reengineering Tools based on ATerms. In Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00), 2000.

[BJW 87] A. Birrell, M. Jones and E. Wobber. A simple and efficient implementation of a small database. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. November 1987, pp. 149-154.

[BKO 98] M. van den Brand, P. Klint and P. Olivier. ATerms: exchanging data between heterogeneous tools for Casl. CoFI Note T-3, 1998.

[BKO 99] M. van den Brand, P. Klint and P. Olivier. Compilation and memory management for ASF+SDF. Technical Report SEN-R9906, CWI-Centrum voor Wiskunde en Informatica, Feb. 28, 1999.

[BKV 96a] M. van den Brand, P. Klint, C. Verhoef. Core Technologies for System Renovation. In K. Jeffery, J. Kral and M. Bartosek (editors), *SOFSEM'96: Theory and Practice of Informatics*, Lecture Notes in Computer Science, Volume 1175, Springer-Verlag, 1996, pp. 235–255.

[BKV 96b] M. van den Brand, P. Klint and C. Verhoef. Reverse Engineering and System Renovation – An Annotated Bibliography.
URL: `http://adam.wins.uva.nl/~x/reeng/REanno.html`

[BLW+ 99] J. Bisbal, D. Lawless, B. Wu and J. Grimson. Legacy Information Systems: Issues and Directions. *IEEE Software*, September/October 1999, 16(5):103-111.

[Bra 98] T. Bray. RDF and metadata, June 1998.
URL: `http://www.xml.com/pub/a/98/06/rdf.html`

[Bra 99] M. Brandel. 1959: The Creation of Cobol. *Computerworld*. March 15, 1999. URL:
`http://www.computerworld.com/cwi/story/0,1199,NAV47_STO34900,00.html`

[Bro 01] G. Brown. COBOL: The Failure That Wasn't. *The Cobol Report*, Object-Z Systems Inc., May 15, 2001.
URL: `www.cobolreport.com/columnists/gary/05152000.htm`

[BS 95] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*, Morgan Kaufmann, San Francisco, 1995.

[BST+ 99] J. Bergey, D. Smith, S. Tilley, N. Weiderman and S. Woods. *Why Reengineering Projects Fail*. Techincal Report CMU/SEI-99-TR-010. Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, April 1999.

[CC 90] E. Chikofsky and J. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*. January 1990.

[CEK+ 00] J. Czeranski, T. Eisenbarth, H.M. Kienle, R. Koschke, E. Plödereder, D. Simon and Y. Zhang. Data Exchange in Bauhaus. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'00)*, Brisbane, Australia, IEEE Computer Society Press, 2000.

[Cha 96] D. Chappell. *Understanding ActiveX™ and OLE*. Microsoft Press, 1996.

[Che 76] P. Chen. The entity relationship model – Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9-36.

[Cor 89] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294-306, 1989.

[Cor 01] J. Cordy. Personal communication. May 2001.

[Cov 98] R. Cover. The XML Cover Pages: XML and semantic transparency. Organization for the Advancement of Structured Information Standards (OASIS), November, 1998.

[Cov 00] R. Cover. The SGML/XML Web Page: Extensible Graph Markup and Modeling Language (XGMML), Organization for the Advancement of Structured Information Standards (OASIS), October 24, 2000.

[CPP 01] CPPX - C/C++ Fact Extractor. Software Architecture Group, The University of Waterloo. URL: `http://swag.uwaterloo.ca/~dean/cppx/`

[CT 99] C. Colket and B. Thomas. *Analysis of Safety-Critical & Mission-Critical Systems Using ASIS*. Presented at the 1999 Software Technology Conference (STC'99).

[daV 98a] *daVinci's Concepts*. daVinci V2.1 Online Documentation. 15-Jun-1998. URL: `http://www.tzi.de/daVinci/docs/general_info/concepts.html`

[daV 98b] *daVinci Term Representation*. daVinci V2.1 Online Documentation. 15-Jun-1998. URL: `http://www.tzi.de/daVinci/docs/reference/api/termrep.html`

[DBI 00] R. Dirckze, D. Baisley and S. Iyengar. XMI – A Model Driven XML Interchange Format. In Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00), 2000.

[DDT 99] S. Demeyer, S. Ducasse and S. Tichelaar. Why FAMIX and not UML? In *Proceedings of UML'99, Lecture Notes in Computer Science, Volume 1723*, Springer-Verlag, 1999.

[DE 94a] P. Devanbu and L. Eaves. *GEN++ - an analyzer generator for c++ programs*. Technical Report, AT & T Bell Labs, NJ, 1994.

[Dev 99] P. Devanbu, GENOA - A Customizable, front-end retargetable Source Code Analysis Framework. *ACM Transactions on Software Engineering and Methodology*, 9(2), April 1999.

[DKV 99] A. van Deursen, P. Klint and C. Verhoef. Research Issues in Software Renovation. In J.-P. Finance (editor) *Fundamental Approaches to Software Engineering (FASE'99), Lecture Notes in Computer Science, Volume 1577*, Springer-Verlag, March 1999, pp. 1-23.

[DMH 01] T. Dean, A. Malton and R. Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, October 2001.

[DRW 96] P. Devanbu, D. Rosenblum and A. Wolf. Generating Testing and Analysis Tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1): 42-62, January 1996.

[EKW 99] J. Ebert, B. Kullbach and A. Winter. GraX - An Interchange Format for Reengineering Tools. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, S. 89-98.

[EKW 00] J. Ebert, B. Kullbach and A. Winter. GraX: Graph Exchange Format. In Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00), 2000.

[Ern 97] J. Ernst. *Introduction to CDIF*. The Electronic Information Group (a division of the Electronic Industries Alliance). September 1997.

[EWD+ 96] J. Ebert, A. Winter, P. Dahm, A Franzke and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In B. Thalheim, (ed.) *Conceptual Modelling – ER'96*, Lecture Notes on Computer Science Volume 1157, Springer, Berlin, 1996, pp. 163-178.

[Fer 01] R. Ferenc. A Short Introduction to the Columbus Proposal for a standard C/C++ Schema. April 17, 2001. URL: `http://www.inf.u-szeged.hu/~ferenc/research/ColumbusSchemaShort.pdf`

[FGS+ 01] R. Ferenc, T. Gyimóthy, S. Sim, R. Holt and R. Koschke. Towards a standard schema for C/C++. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, October 2001.

[FHC 01] H. Fahmy, R. Holt and J. Cordy. Wins and Losses of Algebraic Transformations of Software Architectures, submitted to *Automated Software Engineering (ASE 2001)*, San Diego, California, November 26-29, 2001.

[FH 00a] H. Fahmy and R. Holt. Software Architecture Transformations. In *Proceedings of International Conference of Software Maintenance*, San Jose, California, October 2000.

[FH 00b] H. Fahmy and R. Holt. Using Graph Rewriting to Specify Software Architectural Transformations. In *Proceedings of Automated Software Engineering*, Grenoble, France, September 2000.

[FHK+ 97] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley and K. Wong. The Software Bookshelf. IBM Systems Journal, 36(4), November 1997, pp. 564-593.

[FK 96] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87-94, December 1996.

[Fly 01] P. Flynn (Editor). *The XML FAQ Version 2.01*. June 19, 2001.
URL: `http://www.ucc.ie/xml/`

[FMB+ 01] R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss and M. Tarkiainen. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of the Symposium on Programming Languages and Software Tools (SPLST 2001)*, Szeged, Hungary, June 15-16, 2001, pp. 16-27.

[FV 99] L. Feijs and R. van Ommering. Relation partition algebra – mathematical aspects of uses and part-of relations. *Science of Computer Programming*, 33(2): 163-212, February 1999.

[FW 94] M. Fröhlich and Mattias Werner. *The graph visualization system daVinci – a user interface for applications*. Technical Report 5/94, Department of Computer Science, University of Bremen, 1994.

[GA 95] W. Griswold and D. Atkinson. Managing the Design Tradeoffs for a Program Understanding and Transformation Tool, *Journal of Systems and Software*, 30(1--2):99-116, July-August, 1995.

[GAK 99] G. Guo, J. Atlee and R. Kazman. A Software Architecture Reconstruction Method, *TC2 1st Working IFIP Conference on Software Architecture (WICSA 1)*, February 22-24, 1999, San Antonio Texas, in P. Donohoe. *Software Architecture*, Kluwer Academic Publisher, pp. 15-33.

[GHL+ 92] R. Gray, V. Heuring, S. Levi, A. Sloane and W. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM*, February 1992, pp. 121-131.

[GJS 96] J. Gosling, B. Joy and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GML 01] The GML File Format.
URL: `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/index.html`

[GN 99] E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, John Wiley & Sons, Ltd., 00(S1), 1-5, 1999.

[Gnu 01] GCC Home Page. GNU Project, Free Software Foundation.
URL: `http://gcc.gnu.org/`

[God 99] M.W. Godfrey. Exchange formats: Some problems, a few results, and a cool name. *Presentation at the November 1999 CSER meeting*.

[God 00a] M.W. Godfrey. Defining, Transforming and Exchanging High-Level Schemas. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'00)*, Brisbane, Australia, November 2000.

[God 00b] M.W. Godfrey. High-Level Schemas: A Journey through the Bush. Presented at the *ICSE 2000 Workshop on Standard Exchange Format (WoSEF)*, June 6, 2000.

[God 01] M.W. Godfrey. Practical Data Exchange for Reverse Engineering Frameworks: Some Requirements, Some Experience, Some Headaches. A Position Paper for the ICSE 2000 Workshop on Standard Exchange Format (WoSEF'00). *Software Engineering Notes*, 26(1):50-52, January 2001.

[GW 81] G. Goos and W. Wulf. *Diana Reference Manual*. CMU-CS-81-101, Department of Computer Science, Carnegie-Mellon University, 1981.

[Him 89] M. Himsolt. Graphed: An interactive graph editor. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Vol. 349, Springer-Verlag, Berlin, 1989, pp. 532-533.

[Him 94] M. Himsolt. GraphEd: A Graphical Platform for the Implementation of Graph Algorithms. In R. Tamassia, I.G. Tollis (editors): *Proceedings of Graph Drawing (GD'94), Lecture Notes in Computer Science Volume 894*, Springer-Verlag, Berlin, 1994, pp. 182-193.

[Him 97] M. Himsolt. GML: A Portable Graph File Format. Technical Report. University of Passau, Germany, 1997.
URL: `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html`

[HM 00a] I. Herman and M. S. Marshall. GraphXML – An XML-based graph description format. *Proceedings of the Symposium on Graph Drawing (GD'00)*, Springer Verlag, 2000.

[HM 00b] I. Herman and M. S. Marshall. GraphXML - An XML based graph interchange format. Reports of the Centre for Mathematics and Computer Sciences, Amsterdam, INS-R0009, 2000.

[Hol 97] R. Holt. *Software Bookshelf: Overview and Construction*. Updated 31 Mar 97.
URL: `http://swag.uwaterloo.ca/pbs/papers/bsbuild.html`

[Hol 98a] R. Holt. An Introduction to TA: The Tuple Attribute Language. Department of Computer Science, University of Waterloo and Toronto, November 1998.

[Hol 98b] R. Holt (editor). Conclusions from the Data Exchange Group Meeting at CASCON'98, November 30, 1998.

[Hol 98c] R. Holt (editor). Exchange Formats for Information Extracted from Computer Programs. CSER common data exchange format initiative.

[Hol 98d] R. Holt. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, Honolulu, Hawaii, October 12-14, 1998.

[Hol 00] R. Holt. Software Guinea Pigs. University of Waterloo, 14-April-2000.
URL: `http://plg.uwaterloo.ca/~holt/guinea_pig/`

[HW 00a] R. Holt and A. Winter. A Short Introduction to the GXL Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00), Panel on Reengineering Exchange Formats*, IEEE Computer Society Press, Los Alamitos, CA, 2000.

[HW 00b] R. Holt and A. Winter. GXL: Representing Graph Schemas. Presented at the *7th Working Conference on Reverse Engineering (WCRE'00)*, Brisbane, Australia, 2000.

[HW 00c] R. Holt and A. Winter. Graph exchange Language: An overview. Presented at *CASCON 2000*, Mississauga, Ontario, 2000.

[HWS 00] R. Holt, A. Winter and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00), Panel on Reengineering Exchange Formats*, IEEE Computer Society Press, Los Alamitos, CA, 2000.

[HWS+ 00] R. Holt, A. Winter, A. Schürr and S. Sim. GXL: Towards a Standard Exchange Format. Presentation at the *7th Working Conference on Reverse Engineering (WCRE'00), Panel on Reengineering Exchange Formats*, Brisbane, Australia, 2000.

[IBM 00] XML Metadata Interchange Format, September 1, 2000.
URL: `http://www-4.ibm.com/software/ad/standards/xmi.html`

[ISO 99] ISO/IEC 15291:1999 Information technology – Programming languages – Ada Semantic Interface Specification (ASIS), Edition 1, International Organization for Standardization, 283 pages, Stage Date: 29-Apr-1999.

[Jon 98] C. Jones. The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences. Addison-Wesley, 1998.

[Jor 90] M. Jordan. An extensible programming environment for Modula-3. *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*. December 1990, pp. 66-76.

[JSZ 96] J. Janke, W. Schäfer and A Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In *Proceedings of International Conference on Software Maintenance (ICSM'96)*, IEEE Computer Society, 1996.

[Kam 94] J. Kamperman. GEL, a graph exchange language. Technical Report CS-R9440, CWI, Amsterdam, NL, 1994.

[KCE 00b] H.M. Kienle, J. Czeranski and T. Eisenbarth. The API Perspective of Exchange Formats, In *Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00)*, 2000.

[Kie 01] H. Kienle. Exchange Format Bibliography. *Software Engineering Notes*, 26(1): 56-60, January 2001.

[KGW 98] R. Koschke, J.-F. Girard and M. Würthner. An Intermediate Representation for Integrating Reverse Engineering Analyses. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, IEEE Computer Society, 1998, pp. 241-250.

[KM 94] D. Kinloch and M. Munro. Understanding C Programs Using the Combined C Graph Representation. In *Proceedings of the International Conference on Software Maintenance*, Victoria, BC. IEEE Computer Society Press, 1994, pp. 172-180.

[KN 96] E. Koutsofios and S. North. *Drawing graphs with* `dot`. AT&T Bell Laboratories, Murray Hill, NJ, 1996.

[Kri 97] R. Kirkhaar. Reverse Architecting Approach for Complex Systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, IEEE Computer Society Press, Los Alamitos, pp. 4-11, 1997.

[KSW 95] N. Kiesel, A. Schuerr and B. Westfechtel. GRAS, A Graph-Oriented (Software) Engineering Database System. *Information Systems*, 20(1):21-51, Pergamon Press, 1995.

[KWC 98] R. Kazman, S. Woods and S. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Working Conference on Reverse Engineering (WCRE'98)*, Hawaii, October 1998.

[LA 97] T. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. Computer Science Technical Report TR-97-07, University of Ottawa, November 1997.

[Lam 87] D. Lamb. IDL: Sharing Intermediate Representations. *ACM Transactions Programming Languages and Systems*, July 1987, 9(3): 297-318.

[Lap 00] S. Lapierre. Software Analysis APIs. Presented at the *ICSE 2000 Workshop on Standard Exchange Format (WoSEF)*, June 6, 2000.

[LCH+ 98] T. Lin, R. Cheung, Z. He and K. Smith. Exploration of Data from Modeling and Simulation through Visualization. *Proceedings of the 3rd International SimTect Conference*, Adelaide, Australia, March 1998.

[Lem 98] R. Lemesle. Meta-modeling and modularity: Comparison between MOF, CDIF & sNets formalism. *OOPSLA'98 Workshop #25: Model Engineering, Methods and Tools Integration with CDIF*, October 1998.

[Let 98] T. Lethbridge. Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard. Draft Manuscript, 21-Nov-1998.

[Let 01] T. Lethbridge. The Dagstuhl Middle Model (DMM), Version 0.003, 06-Jun-2001.

[LL 00] C. LeDuc and B. Lague. Datrix Abstract Semantic Graph Reference Manual Version 1.4, Bell Canada, May 1, 2000.

[LLL 01] S. Lapierre, B. Laguë and C. Leduc. Datrix Source Code Model and its Interchange Format: Lessons Learned and Considerations for Future Work. *Software Engineering Notes*, 26(1): 53-56, January 2001.

[LS 97] T. Lethbridge and J. Singer. Understanding Software Maintenance Tools: Some Empirical Research. Submitted to *Workshop on Empirical Studies of Software Maintenance (WESS'97)*, Bari, Italy, October 1997.

[Mar 99] J. Martin. *RSF File Format*. Posted to the Rigi Developer Email Distribution List, August 19, 1999.

[Mic 99] Microsoft Corporation. Repository SDK 2.1b Documentation: XML Interchange Format (XIF), Redmond, WA, May 1999.

[MJS+ 00] H. Müller, J. Jahnke, D. Smith, M. Storey, S. Tilley and K. Wong. Reverse Engineering: A Roadmap. In *A. Finkelstein (ed.) Future of Software Engineering, International Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, June 2000.

[MOT+ 93] H. Müller, M. Orgun, S. Tilley and J. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*. 5(4):181-204, December 1993.

[MRE 01] The Mozilla Reverse Engineering Project. Department of Computer Science, University of Victoria.
URL: http://www.rigi.csc.uvic.ca/Pages/projects/mozilla.html

[Mül 98] H.A. Müller, Criteria for Success of an Exchange Format. Workshop meeting minutes, CASCON'98. November 30, 1998.

[MWT 94] H. Müller, K. Wong and S. Tilley. Understanding software systems using reverse engineering technology. In Proceedings of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS 1994).

[MV 92] A. von Mayrhauser and A. M. Vans. *An industrial experience with an integrated code comprehension model.* Technical Report CS-92-205, Colorado State University, 1992.

[Nag 96] M. Nagl (Editor). Building Tightly Integrated Software Development Environments: The IPSEN Approach. Springer-Verlag, Berlin Heidelberg, 1996.

[OMG 91] *Common Object Request Broker Architecture and Specification*. Document Number 91.12.1, Object Management Group.

[OMG 97] The common request broker: Architecture and specification, revision 2.0. Technical Report 97-02-25, Object Management Group, 1997.

[OMG 98] XML Metadata Interchange (XMI), Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF), Document Number ad/98-10-05, Object Management Group, October 20, 1998.

[OMG 01a] *Unified Modeling Language Specification Version 1.4*. Object Management Group, September 2001.

[OMG 01b] *UML 2.0 Diagram Interchange Request For Proposal*. Object Management Group, 06-Apr-2001.

[OSF 91] Open Systems Foundation. OSF Architecture-Neutral Distribution Format Rationale, June 1991.

[PBS 01] PBS Toolkit – Reference. URL: http://swag.uwaterloo.ca/pbs/

[Per 00] S. Perelgut. The Case for a Single Data Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, IEEE Press, 2000.

[PKS 98] P. Pagé, R. Keller and R. Schauer. A JavaCC parser for the UML-based CDIF transfer format. In Workshop on Model Engineering, Methods and Tools Integration with CDIF, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), Vancouver, B.C., Canada, October 1998.

[PT 90] F. Paulish and W. Tichy. Edge: An extensible graph editor. *Software – Practice and Experience*, 20(S1):S1/63-S1/88, June 1990.

[Rat 97] Rational Software. *UML-Compliant Interchange Format*. OMG document ad/97-01-13, 1997.

[Rek 85] M. Rekoff, Jr. On Reverse Engineering. *IEEE Transactions on Systems, Man and Cybernetics*. March/April 1985, pp. 244-252.

[RF 00] S. Rybin and V. Fofanov. *Building ADA Development Tools with ASIS-for-GNAT*. Presented at SIGAda 2000.

[Riv 00a] C. Riva. Reverse Architecting: Suggestions for an Exchange Format. In Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00), 2000.

[Riv 00b] N. Rivard. The UML-Xchange Home Page.
URL: `http://www.cam.org/~nrivard/uml/umlxchng.html`

[RW 91] D. Rosenblum and A. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *Proccedings of the USENIX C++ Conference*, Washington, DC, April 22-25, 1991.

[RW 96] S. Rugaber and L. Wills. *Creating a research infrastructure for reengineering*. In Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96), IEEE Computer Society Press, Los Alamitos, 1996, pp. 98-102.

[San 95] Georg Sander. *VCG: Visualization of Compiler Graphs*. Technical Report A01-95, Universität des Saarlandes, FB 14 Informatik, February 1995.

[SB 94] E. Schonberg and B. Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of Tri-Ada'94*, Baltimore, Maryland, ACM Press, 1994, pp. 48-57.

[Sch 97a] A. Schürr. Developing Graphical (Software Engineering) Tools with PROGRES, Formal Demonstration. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, IEEE Computer Society Press, Los Alamitos, May 1997, pp. 618-619.

[Sch 97b] A. Schürr. *Progres for Beginners*. Department of Computer Science, Aachen University of Technology, 1997.

[Sch 00] A. Schürr. *Looking for a Graph eXchange Language*, presentation at the APPLIGRAPH - Subgroup Meeting on Exchange Formats for Graph Transformation, Paderborn University, Germany, September 5 - 6, 2000.

[SFM 97] M. Storey, D. Fracchia and H. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization. In *Proceedings of the 5th Workshop on Program Comprehensio*n. Dearborn, Michigan: May 28-30, 1997. IEEE Computer Society Press, 1997, pp. 17-28.

[SHK 00b] S. Sim, R. Holt and R. Koschke. WoSEF – Workgroup on Standard Exchange Format, Progress Towards a Format.
URL: `http://www.cs.toronto.edu/~simsuz/wosef/`

[Sim 00a] S. Sim. Next Generation Data Interchange: Tool-to-Tool Application Program Interfaces. In Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'2000), Brisbane, Queensland, Australia, November 2000, pp. 278-283.

[Sim 00b] S. Sim. Rogues Gallery of Schemata, 13-Nov-2000.
URL: `http://www.cs.toronto.edu/~simsuz/schema/`

[Sim 00c] S. Sim. WoSEF Events.
URL: `http://www.cs.toronto.edu/~simsuz/wosef/events.html`

[Som 01] I. Sommerville. *Software Engineering, 6th Edition*. Addison-Wesley, New York, 2001.

[SSK 00] G. St-Denis, R. Schauer and R.K. Keller. Selecting a Model Interchange Format: The SPOOL Case Study, *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, IEEE Press, 2000.

[SWF+ 96] M. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins and H. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proceedings of the 3rd Working Conference on Reverse Engineerin*g *(WCRE'96)*. Monterey, CA: November 8-10, 1996. IEEE Computer Society Press, 1996, pp. 31-40.

[SWM 97] M. Storey, K. Wong and H. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*. Amsterdam, The Netherlands: October 6-8, 1997. IEEE Computer Society Press, 1997, pp. 12-21.

[SY 98a] J. Suzuki and Y. Yamamoto. Toward the Interoperable Software Design Models: Quartet of UML, XML, DOM and CORBA. Proceedings of the *4th IEEE International Symposium and Forum on Software Engineering Standards*, IEEE Computer Society Press, 1998.

[SY 98b] J. Suzuki and Y. Yamamoto. Managing the software design documents with XML. In *Proceedings of the 16th International Conference on Systems Documentation, Working with XML*, ACM Press, 1998, pp. 127-136.

[TDD 00] S. Tichelaar, S. Ducasse and S. Demeyer. FAMIX: Exchange Experiences with CDIF and XMI. In Proceedings of the Workshop on Standard Exchange Formats (WoSEF) at the 22nd International Conference on Software Engineering (ICSE'00), 2000.

[Til 98] S. Tilley. *Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities in 1998*. Technical Report CMU/SEI-98-TR-001, Software Engineering Institute, Carneigie Mellon University, February 1998.

[TPS 96] S. Tilley, S. Paul and D. Smith. Towards a Framework for Program Understanding. In *Proceedings of the 4th International Workshop on Program Comprehension (IWPC)*. March, 1996, Berlin, Germany, pp. 19-28.

[TS 97] S. Tilley and D. Smith. On Using the Web as Infrastructure for Reengineering. In *Proceedings of the 5th Workshop on Program Comprehension (IWPC'97)*. May 1997, Dearborn, MI, pp. 170-173.

[TSC 98] Tom Sawyer Software Corporation. *Graph Layout Toolkit User's Guide*. 1998. URL: `http://www.tomsawyer.com/manuals/get30/gltusers/gltusers.htm`

[Ulr 01] W. Ulrich. Remember Cobol? If You Don't, Get Reacquainted. *Computerworld*. May 21, 2001.
URL: `www.computerworld.com/cwi/story/0,1199,NAV47_STO60683,00.html`

[WCK 99] S. Woods, S. Carrière and R. Kazman, A semantic foundation for architectural reengineering and interchange. *Proceeding of the 1999 International Conference on Software Maintenance (ICSM '99)*, IEEE Computer Society Press, Oxford, England, August 1999, pp. 391-398.

[Wei 01] B. Weinstein. COBOL programmers are back in the game. *Tech Watch*, Tech Republic, Inc., March 7, 2001.
URL: `http://www.cobolwebler.com/cobol_program.htm`

[Wil 01] S. Wilkinson. From the dustbin, COBOL rises: Long-lived legacy apps keep creaky skills crucial. *EWeek*, Ziff Davis Media, May 28, 2001. URL:
`http://www.zdnet.com/eweek/stories/general/0,11011,2764006,00.html`

[Win 01] A. Winter. *GXL: Graph Exchange Language*. Presented at Dagstuhl Seminar 01041: Interoperability of Reengineering Tools, Dagstuhl, Saarland, Germany, January 21-26, 2001.

[WOL+ 98] S. Woods, L. O'Brien, T. Lin, K. Gallager and A. Quilici. An architecture for interoperable program understanding tools. In *Proceedings of the 6th International Workshop Program Comprehension (IWPC'98)*, 1998, pp. 54-63.

[Won 98] K. Wong. *RIGI User's Manual – Version 5.4.4*. Department of Computer Science, University of Victoria, June 1998.

[WWW 99] *HTML 4.01 Specification*. World Wide Web Consortium Recommendation, December 24, 1999.

[WWW 00] *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium Recommendation, October 6, 2000.

[ZK 01] Y. Zou and K. Kontogiannis. Towards a Portable XML-based Source Code Representation. In *Proceedings of ICSE 2001 Workshops of XML Technologies and Software Engineering (XSE 2001)*, Toronto, May 15, 2001.