Defining Re-engineering

The most quoted definition of software re-engineering is “the examination and alteration of a software system to reconstitute it in a new form and the subsequent implementation of the new form” [Chikofsky 1990]. Another cogent definition of re-engineering is “the process of creating an abstract description of a system, reason about a change at the higher abstraction level, and then re-implement the system. […] Re-engineering = Reverse engineering + Δ + Forward Engineering” [Jacobson 1991].

Changing information at the same level of abstraction is commonly referred to as restructuring, i.e., “the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. While restructuring creates new versions that implement or propose change to the subject system, it does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system” [Chikofsky 1990].

Re-engineering typically involves re-structuring during the abstraction stage
- Re-structuring can be used as a preparation step for re-engineering a system, such that it is understood by the reverse engineering tool, or by reducing dependencies between sub-systems, such that only portions of the system are re-engineered (incremental re-engineering)
- Re-engineering also involves re-structuring during the final refinement stage, when it is time to semi-automatically reconstitute the software in its new form.

In the context of object-oriented systems, refactoring is more commonly used term: “the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” [Opdyke 1992]. One may argue that checking for behavior preservation after refactoring is easier said than done, and that in some cases, relaxing this constraint is more appropriate. For instance, replacing conditionals with polymorphism may result in a net performance improvement!

An interesting distinction between restructuring or re-engineering is articulated in [Colbrook 1990]: restructuring only alters the “Surface Plan” as termed by Rich and Strobe in their Programmer’s Apprentice work at MIT, i.e., the outward control flow and appearance of the code, and not its underlying data representations and structures. Whereas re-engineering affects the selection of the data representations and their structures, or in general, the coarser grains of abstraction, manifested as the “Deep Plan”, which greatly determines the elegance, simplicity

---

![Figure 1: Re-engineering as abstraction, transformation, and reimplementation [Chikofsky 1990]](image-url)
and maintainability of the final code.

To recapitulate, the re-engineering paradigm can be summarized as abstraction, transformation, and reimplementation [Waters 1988]. This paradigm is conceptualized by [Byrne 1992] in Figure 2.

### The Case for Re-engineering

A legacy system is defined as one that “significantly resists modification and evolution to meet new and constantly changing business requirements”, regardless of the technology from which it is built [Brodie 1995]. In 1994, a survey by IBM’s Center for Advanced Studies found that the average Fortune 100 Company maintains 35 million lines of code and adds an additional 10 percent each year just in enhancements, updates, and normal maintenance. As a result of maintenance alone, software inventories will double in size every seven years [Buss 1994]. These valuable systems will not be replaced overnight, yet many are exhibiting the characteristics of legacy systems.

Reusing requirements and business logic already represented in legacy systems can be invaluable [Arnold 1986]; it could be as trivial as the size of a database field, or as complicated as a component which encodes complex rules. [Britcher 1990] eloquently describe some advantages of re-engineering legacy systems: “reengineering avoids 1) the effort of, and the errors that accrue from, developing new engineering requirements and functional specifications, and 2) the effort of full-scale verification and validation[…] In short, re-engineering takes advantage of the profound effects of evolution. It preserves the functional behavior of a system that had been specific, designed, implemented, repaired, enhanced, verified, validated, and most importantly, used over years, while improving its quality.” The internal value of re-engineering is then the cost savings owing to extending the lifetime of a system, and delaying the introducing of a new system built from scratch.

Determining when and how to re-engineer a legacy system\(^1\) is mainly a business decision to preserve or increase the value of existing software investments (see Figure 1). [Stevens 1998] notes that many corporations lack the re-engineering expertise. The research literature contains various experience reports involving actual re-engineering projects [Sneed]. In some cases, even if successful, the results do not entirely generalize when re-engineering is done manually, mainly capitalizing on modern software practices [Britcher 1986], [Jacobson 1991], [Markosian 1994], [Gannod 1998].

A short article by [Ulrich 1988] has an eloquent argument for re-engineering: “re-engineering is not shorthand terminology for reverse engineering but is, in and of itself, a viable, useful method of improving systems […] re-engineering is the first, essential step toward reverse engineering. It begins ”cleaning up the house. [...] While reverse engineering sounds like the panacea for extending the life of existing business systems, there's just one catch. The automated reverse engineering piece of the puzzle--the one that takes a current business system and extracts design specifications from it [so that it can be re-implemented in modern CASE tools] doesn't exist today. So, what do you do to position yourself for the future? [...] you can re-engineer your systems now.”

\(^1\) Software re-engineering is not to be confused with business process re-engineering [Champy 1993].
This perspective clarifies the goal or standard of progress. Re-engineering is not so concerned with obtaining a complete specification of an existing system as in reverse engineering [Rekoff 1985]; re-engineering can be more approximate and require sufficient information to perform the desired alterations — some of which can be derived automatically from the underlying representation, other has to be specified by the user. Re-covering missing design information is the key element which makes software maintenance hard. [Biggerstaff 1989] was one of the first to introduce design recovery and note that any design recovery must also capture and express design abstractions that are not explicitly represented in code, possibly sacrificing formal completeness for scalability.

Re-engineering can be explained as a coping mechanism for the lack of transformational programming, as advocated by [Balzer, 1976], which would enable maintenance by modification of specification and reimplementation. Furthermore, the timeline of the evolution of re-engineering technologies shows higher and coarser levels of abstractions, and is closely tied to the evolution of various techniques and formalisms for coarser data abstraction, from Abstract Data Types (ADTs) to Module Interconnection Languages (MILs) [DeRemer 1976] and Architecture Description Languages (ADLs) [Medvidovic 2000].

**Evolution of Re-engineering**

[Jacobson 1991] classifies the possible transformations into two orthogonal dimensions: change of functionality, and change of implementation technique. One can also classify the alteration of the system as manual, semi-automated, or fully automated. The early code restructuring tools dealing with millions of lines of code of legacy code tended to be batch oriented with mixed results. Re-engineering at higher-levels of abstraction cannot be fully automated as such knowledge cannot be completely derived from the existing implementation. At higher-levels of abstraction, the approaches tend to be semi-automated and more interactive, with a heavy emphasis placed on direct manipulation and visualization. Generally, identifying the kind of change to make is still mostly manual, with the tool automating the propagation of the local change [Griswold]. As a concrete example, in a modern IDE with support for refactoring, selecting an `ExtractMethod` refactoring prompts the user for the name of the method to generate and does not generate a cryptic method name; even if the tool suggests names for the method parameters based on their declared types, the user can always override the automatically generated names.

In the remainder of this discussion, we classify the changes into the following dimensions:

- Re-engineering Unstructured Legacy Code
- Re-engineering Legacy Databases
- Re-engineering Abstract Data Types
- Re-engineering Modules
- Re-engineering Designs
- Re-engineering Architectures

For each of the above sub-areas, this report will briefly survey the evolution of the technology and the various notations, formalisms and techniques to support re-engineering, including any tool support. We’ll also mention any additional research methods that have been used successfully, such as case studies and experience reports.

**Re-Engineering Unstructured Legacy Code**

By the time structured design techniques were popularized [Stevens 1974], companies were faced with millions of lines of legacy code and experiencing high maintenance costs. One can find a few controlled experiments to measure programmers’ perceptions of improvements due to re-engineering or restructuring a legacy system [Gibson 1989].
Technology Evolution

The earliest re-engineering work attempted to impose block structure on poorly structured legacy code. Automatic restructuring systems were developed as early as the mid-1970s to restructure legacy COBOL or FORTRAN programs. [Olsem 1995b] lists quite a few tools that are available.

Same Target Language. [de Balbine 1975] is an early approach to remove gotos from programs. [Baker 1977] proposes an algorithm for structuring flowgraphs. [Calliss 1988] notes many problems with automatic restructurers, such as generating cryptic variable names, and leaving behind misleading comments.

Different Target Language. Re-structuring legacy code to a different target language is more restricted. Re-coding at the same level of abstraction cannot effectively take advantage of the richer constructs in the new language. [Olsem 1995] notes that line-for-line translations from COBOL to Ada are derisively called “Adabol”. [Baxter 2004] shows some of the newer techniques available.

Clone Detection and Elimination. By some estimates, over 10% of code bases may consist of duplication. When legacy systems become hard to maintain, reuse by copy-paste-modify is a common coping mechanism used by developers. Various approaches have been proposed [Baxter 1998] use a program graph comparison over Abstract Syntax Trees.

Eliminating Non-Traversable Paths. [Pleszkoch 1992] proposes an approach to eliminate non-traversable path, which are perceived as an impediment to maintainability.

Techniques and Formalisms

Augmented Abstract Syntax Tree. This type of re-engineering relies on a parse tree with additional data and control flow information, and applies transformation rules on the AST to generate a new version of the original system. Some commonly used techniques are:
- Control Flow Graph (CFG): used to reorder statements within a program
- Program Dependency Graph (PDG): used for restructuring [Griswold 1993a], clone detection
- Unified Inter-Procedural Graph (UGI): [Harold 1993]
- Context Entity Graph (CEG): places its main emphasis is on generating new source [Eloff 2001]

Re-engineering Legacy Databases

As reported in [Cohen 2003], Gartner estimates there are about 15,000 installations of pre-relational databases, such as indexed-sequential, hierarchical, and network databases, and about 50% would consider reengineering to relational databases.

Technology Evolution

Re-engineering databases seemed to predate re-engineering software. One of the early pioneers [Warnier 1974], advocated changing the program structure to closely mirror the data it accesses. [Mens 2004] claims that the research area of object-oriented software refactoring originated in the research of restructuring object-oriented database schemas. The problem of view updates in databases inspired many approaches to model transformation, and checking for consistency between views.

Techniques and Formalisms

Distinguishing between Types and Instances. [Navathe 1976] is one of the earliest papers to addresses schema translation. It identified the interaction between schema and instance levels. It identified a first level of abstraction as being the schema, and a set of restructuring operations for schemas: naming, combining and relating. A second level of abstraction as being the instance operations, and the corresponding transformations on the data instances, such as replication, factoring and union. Finally, the third-level of abstraction affects item value operations, such as copying, deleting, etc.
**Code Translation.** [Cohen 2003] describes the more general database-software translation problem consisting of schema translation, data migration, and code translation, and proposes an automated technique to transform an original program in a high-level language (e.g., COBOL) with embedded custom database access commands into a semantically-equivalent program in the same host language but using embedded SQL for database access. Using the paradigm of translation by abstraction, transformation, and reimplementation, described by Waters [1988]: the original program is first analyzed into a more abstract representation describing database operations, abstracting away from syntactic coding variations. The abstract representation is then queried for common database access patterns and transformed into a form closer to the target language. Finally, the abstract representation of the program is re-implemented in the target language as SQL queries.

**Re-engineering Abstract Data Types**

Early work in program transformation was concerned with changing the abstract data types.

**Technology Evolution**

[Bowdidge 1995] classifies transformations into four categories:

- **Scoping transformations:** change where functions and variables are defined and visible
- **Syntactic transformations:** perform superficial changes to the program’s parsed form (e.g., rename a variable)
- **Control flow transformations:** affect the order in which statements occur.
- **Abstraction transformations:** create and destroy named objects in the code.

**Techniques and Formalisms**

**Program Slicing.** [Weiser 1984] introduced slicing, i.e., identify the elements that can be affected by an element.

**Cliché-based techniques.** [Rich 1990] illustrate how translating programs into the Plan Calculus helps recognizing clichés and overcomes the difficulties of syntactic variation and non-contiguosness by abstracting away from the details of algorithms that depend only on their expression in code. Cliché-based techniques can be classified as automated, semi-automated, or manual. These bottom-up techniques suffer from problems of scale, and may produce views that are too detailed. Furthermore, the need to identify the knowledge base of clichés. However, unlike the reflexion model technique, cliché-based approaches can potentially capture both behavioral and structural information.

**Program transformation.** [Arango 1986] is one of a series of papers which propose a transformation based maintenance model. The approach was used to port a program completely automatically from one computing environment to another. A maintenance program is represented using a DAG, where the root represents the initial system specification, and leaves are the programs. Tactics control navigation through the graph, and maintain the design rationale. In order to effect a transformation, i.e., reverse a design decision, the least common abstraction is found: it is a node along the path up the refinement DAG toward the root, that encompasses both current and desired implementation. E.g., if the design decision was to use a HeapSort algorithm, one can go back and change the design to use a QuickSort algorithm. The function is changed by changing the specification, and the new specification is refined to a particular implementation. The authors offer an insight into the reason why maintenance degrades the structure of a program: each maintenance activity introduces cumulative approximation errors, i.e., difference between the maintenance DAG and the actual DAG.

**Restructuring Tool Support.** [Griswold 1992] recognized that manual restructuring is error prone, because local changes need to be propagated non-locally, and if not done correctly, can introduce bugs into a previously working program. [Griswold 1993] proposed an approach for restructuring programs, using a set of 20 basic transformations. The programmer is responsible for manually selecting the restructuring transformation to apply. The tool propagates the local change.
Star Diagrams. An interesting research question was then finding ways to help programmers perform composite transformation, such as perform encapsulation, with a restructuring tool, i.e., how to support program restructuring through program representations other than the source code. [Bowdidge 1995] created the star diagram program visualization technique to help a programmer encapsulate global variables or data structures into a new abstract data type. The diagram shows how the data structure is used throughout the program, hiding the irrelevant portions of the program and presenting the code directly related to the data structure. The view groups similar expressions together in order to identify common abstract operations that should become the functions forming the interface of the ADT. The view supports direct manipulation, which cause meaning-preserving restructuring transformations to be applied to the underlying program. [Bowdidge 1998] provides the clearest and deepest description of the star diagram concept. The predecessor to the Star Diagram was the Structure Diagram described in a workshop paper [Griswold 1993b], a program representation that showed the named objects of the program (functions, variables, and modules), represented their relationships, and restructured the source code as the structure diagram was manipulated. The supported restructurings included extract function, extract parameter, etc...The program performs syntactic and semantic checks using an Abstract Syntax Tree and a Program Dependence Graph to ensure that the transformation is meaning-preserving. The technique is applied to automatically restructure Parnas’s KWIC program from a functional decomposition into a data decomposition.

The Star Diagram was later applied to C [Griswold, 1996], and a star diagram re-engineering tool for Java (Elbereth) [Korman 1998], [Hayes 2000] tried to make the approach and tool support able to support multi-language star diagrams. Uniformly handling the abstract syntax trees (AST's) of different languages and showing star diagrams that contain code fragments from different languages will enable re-engineering legacy systems consisting of a mix of languages, a common situation. Being able to handle multiple languages is a recurring idea with more mature approaches, such as [Baxter 2004]. Re-engineering some of the modern code bases which support platform neutrality will make this an absolute requirement: e.g., in Microsoft.NET, a developer can write a C# class extending a VB.NET class, and overrides methods. Similarly, a COM component can be used from Visual C++ or Visual Basic 6, requiring changes to both.

Empirical Study
[Griswold 1998] and [Bowdidge 1997] reports interesting results on a usability study where developers actually evaluated the Star Diagram technique to perform re-engineering tasks. [Bowdidge 1995] evaluates the scalability of star diagrams on C programs ranging from 100,000 to 500,000 lines, and compares the approach to using UNIX tools to perform restructurings. The studies pointed out a couple of interesting results: restructuring tools encouraged programmers to perform less up-front design planning; and programmers lack cues to determine the “goodness” of a given transformation. These recurring themes still exist today: Extreme Programming encourages less up-front design, and does not rely on objective metrics to detect code in need of refactoring.

Re-engineering Modules
Parnas made a good argument for having a good modular decomposition based on the principle of information hiding, advocating that a module should be “... characterized by a design decision which it hides from all others. Its interface or definition [is] chosen to reveal as little as possible about its inner workings”.

Schwanke observed that “without an independently-specified system architecture, modularity frequently deteriorates over the lifetime of a system. Each time that a programmer adds a new procedure to the system, he must decide which existing module he should place it in. Sometimes the correct decision should be to form a new module, containing this procedure and procedures drawn from existing modules, but the mental and administrative effort involved often deters him. Either way, the programmer often has only a worm’s eye view of the system, from the corner where he is working, and makes his organizational decisions accordingly. Sooner or later, someone on the project usually notices that the organization has deteriorated. Then, a small team of experts is appointed as “architects”, to analyze and reorganize the system. However, their task is even more formidable than the individual programmer’s, because they must understand many more system-wide interrelationships, and must carry out widespread changes without
causing the system to break. Furthermore, when the programming language and tools do not support modularity adequately, they must analyze actual cross-reference information to deduce the scopes of many program units, rather than relying on specifications.” [Schwanke, 1991]

Some programming languages have first class module constructs, making it easier to determine the module structures from the code. For such programming languages, even with weak module constructs, e.g., Java, restructuring the module view of an existing implementation seems to have reached the popularization stage, with commercially available tools, e.g., [Compuware], [Hautus 2002] for Java. In other cases, the majority of the techniques are still in the exploration stage; most approaches are based on clustering and slicing driving manual restructuring.

**Technology Evolution**

Researchers very early on recognized that partitioning interconnected program modules into clusters can reduce complexity. In [Belady 1981], the main parameters are the number of clusters to be generated and the maximum number of nodes allowed in each cluster. The authors also proposed a measure of complexity, based on the intuition that understanding or manipulating interconnected elements is more difficult if their number is large (spread, i.e., count of elements under consideration). And that given the elements, complexity is proportional to the number of connections (richness of connectivity).

[Calliss 1990] attempts to detect potpourri modules, i.e., modules that provides more than one service to a program, violating the principle of a module decomposition advocated by Parnas, that of “responsibility assignment”. Graph manipulation operations are used to perform inter-module analyses: for a given module, analyze its entity-to-entity interconnection graph to find any proper subgraphs. Module factoring is then used to split a module into smaller and more cohesive modules.

[Ammann 1994] describes a radical solution targeted towards module manipulation “in-the-large” with tool support to propagate changes to all affected modules: an inter-module organizer automates the task of moving program entities between modules, such that the import/export declarations are properly updated. A renamer supports renaming with capture avoiding substitution. These tools offer higher level of modifications that the ones normally performed in text editors. This approach also illustrates the notion of transactional refactoring described by [Tokuda 2001], where an individual refactoring might be incorrect, but a compound transformation is correct.

[Quilici 1995] recognized the fully automated extraction of high-level information is not realistic and proposed using cooperative extraction of the remaining specification, wondering whether or not this approach will scale in practice to real world legacy systems.

Around the same time, [Murphy 1995] proposed Reflexion Models, which was successfully used to re-engineer a one-million line of code program [Murphy 1997a]. This radical tool is a lightweight, approximate, semi-automated approach to check structural conformance between a high-level model and a source-level model. It is lightweight, approximate, and scalable. However, it is restricted to structural conformance. Cliché-based approaches can also check for behavior. Hierarchical Reflexion Models were proposed more recently [Koschke 2003] to be able to scale more effectively on industrial-sized projects.

**Techniques and Formalisms**

**Software Metrics**

The main idea behind using software metrics for re-engineering as follows [Arnold 1986]:

1. measure the software with the software metric
2. from the metric’s value, answer the question: “Is the software property measured by the metric satisfactory?”;
3. if not, re-engineer or re-structure the software and go to step 1;
4. if so, you’re done.
[Welker 1994] reports a case study where metrics were tracked during a re-engineering project to come up with a maintainability index, and noted that the re-engineered software had a substantially higher maintainability index, using similar measures that were developed at Hewlett Packard [Coleman 1994].

Case Studies

[Lange 1991] describes one of the early modularization tools, Arch, developed at Siemens Research on an actual case study.

[Murphy 1997a] reports how the Reflexion Models technique was used by an engineer at Microsoft to perform an experimental reengineering of the million line-of-code Excel code base. The case study led to an interesting observation, that in some cases, the re-engineer relied heavily on the tool’s textual output, as a direct visualization of the source or extracted models such as call graphs may not be feasible.

[Dayani-Fard 2005] reports on a case study on how restructuring legacy C/C++ software significantly reduced the build time, which is an important consideration for the s

Re-engineering Designs

The technology maturation for re-engineering design views seems to have occurred with the simultaneous appearance of object-oriented refactoring (1992), design patterns (1993), and semi-formal notations such as UML (first officially released in 1997).

Technology Evolution

Refactoring ([Opdyke 1992], [Moore 1996], [Fowler 2000], [Mens 2004]) is at the beginning of the Popularization stage in the Redwine-Riddle maturation model [Redwine 1985]. [Roberts 1997] proposed a refactoring browser for Smalltalk. Refactoring is now available in most modern Integrated Development Environments [IDE]. Today’s refactoring tool support performs only syntactic checks. Unlike earlier attempts at checking for behavior preservation, the burden of proof that a refactoring did not introduce any new bugs rests with the developer. The Extreme Programming trend, with its “test first” approach, strongly advocates using refactoring only when unit tests [JUnit] are available for equivalence testing [Eloff 2002]. Little guidance is provided for identifying what refactorings are needed, although some work is being done to assist the developer in identifying refactoring opportunities [Tourwé 2003].

During the same time, UML was being standardized, and resulted in several tools with code generation capabilities. Many commercial UML tools are available to automatically retrieve the static structure (e.g., the class diagram) of an object-oriented system. [Briand 2003] notes some of the remaining challenges with distinguishing between association, aggregation and composition relationships, and the reverse engineering of many-to-many associations. Some of these may require semantic or static analysis of the source code. Retrieving the dynamic structure did not occur until recently. Also, it is not until recently that direct manipulation tools became available. Some promising approaches enforce the consistency between various design-level artifacts, such as class-diagrams, activity diagrams and state transition diagrams [Boger 2002]. A strongly related area is finding differences and merging UML models [Alanen 2003].

Design patterns were popularized in [Gamma 1993]. [Budinsky 1996] illustrates how design patterns can be used for code-generation. A design view can be re-engineered to introduce a design pattern [OCinneeide 1999], or check the correctness of the implementation of a design pattern [Nickel 2000].

Techniques and Formalisms

The main techniques and formalisms used here include:

2 I could not determine if the first tool to reverse engineering UML class diagrams started out as a research prototype or in industry. Today, there are too many commercial offerings to list here.
**Invariants, Pre and Postconditions.** In the context of object-oriented refactoring, Opdyke proposed using preconditions as enabling conditions for each refactoring, and using the preconditions to preserve invariants [Opdyke 1992].

**Correctness Checks.** Various checks have been proposed to check for behavior preservation. Access preservation means that all variable accesses should be preserved by the refactoring. Update preservation means that all variable updates must be preserved by the refactoring. Conventional type systems can also check for behavior preservation by enforcing that types are not changed (modulo subtyping) by the refactoring.

**Graph Transformations.** Graph transformations can strengthen diagrammatic models by formalizing the semantics, and thus provide a strong basis for reasoning on diagrammatic models at all levels. However, graph transformations do not scale well [Baresi 2005]. An important element is that graph transformations require the presence of type graphs or metamodels to evolve existing models. Graph transformations have been successfully used in a round-trip engineering environment, FUJABA [FUJABA], [Nickel 2000], to detect the presence of design patterns in code.

**Software Metrics**

Some attempts have been made at using structural metrics to detect parts of the code in need of restructuring [Steinbrückner 2001] [Tourwé 2003]. The hardcore Extreme Programming community has not adopted them yet: “In our experience, no set of metrics rivals informed human intuition” [Fowler 1999, p. 75].

**Re-engineering Architectures**

Re-engineering at the architectural level is a nascent research area, and significantly complicated by the need to abstract an architectural view of a legacy system [Harris 1995] [Kazman 1998]. There is a recent trend to specify an architectural component-and-connector (C&C) view directly in code [Aldrich 2002], making it easier to retrieve such a C&C view from the code.

**Technology Evolution**

[Erdogmus 1998] attempts to classify the various kinds of architectural evolution, such as Replacement and Extension. The architectural evolution paths with deltas he describes are very similar to the Software Maintenance by Transformation (TMM) approach [Arango 1988].

[Krikhaar 1999] illustrates the process: in the architecture impact analysis phase (Phase I), structural changes are made to the architectural model to determine the resulting architecture. If the resulting architecture is desirable, the structural changes are applied to the system to obtain the new architecture, during a transformation phase.

**Techniques and Formalisms**

**Architecture Description Languages.** [Aldrich 2002a] [Aldrich 2002b] contain case studies describing how existing applications were re-engineered using an implementation-oriented Architecture Description Language (ADL), ArchJava, which embeds an architectural specification directly within code. Similar techniques could be used with implementation-constraining ADLs with code generation capabilities or implementation independent ADLs such as C2 [Medvidovic 00] that provide an implementation framework for code generation.

**Graph Transformations.** Architectural styles have been modeled using Graph Transformation [Le Metayer] and [Hirsch 1998]. However, there are other ways of representing architectural styles; and it is not clear if those representations are amenable to such transformations.
**Algebraic Manipulations.** [Holt 1999] and [Fahmy 2001] illustrate the successes and limitations of using algebraic manipulations to abstract and aggregate the views obtained during architectural reconstruction. Specifying the manipulations in such a formalism holds the promise of being directly executable.

**Cliché-based approaches.** [Fiutem 1996]

**Slicing.** [Zhao, 2002] introduced the use of slicing for software architectures.

**Software Metrics**
[Henry 1981] proposed metrics for evaluating the structure of large scale systems based on information flow and correlates the complexity measure with the occurrence of changes and structural flaws.

**Case Studies**
Several case studies involving repairing the architecture of existing systems have been documented. However, these case studies do not exhibit good research maturity. For instance, the developers did not adopt the recommended changes by the researchers. [Tran 1999] describe forward repair, as altering the extracted architecture (or concrete architecture) to be more consistent with the mental model of the software. And reverse repair, as altering the conceptual architecture to be more consistent with the concrete architecture.

**General Approach to Re-engineering**
Many of the approaches surveyed can be expressed in terms of a graph-based approach. Information at one level of abstraction can be represented as a source graph. Information at another level of abstraction can be represented as a target graph. A correspondence graph establishes mapping between subgraphs of source and target graph: a correspondence node is connected to the nodes of these subgraphs by correspondence edges. Correspondence nodes may be mutually connected by edges representing dependencies between mapping decisions, e.g., mapping an UML class to a Java class results in the mapping of composed objects of the UML class (attributes and operations) to elements of the corresponding Java class, within the scope of the class mapping [Hausmann 2003].

For maximum generality, one has to use hierarchical or nested graphs. Trees can be used in certain restricted domains; e.g., if one makes the assumption that an element cannot be part of more than one module. Also, for maximum generality, there may not be unique identifiers associated with each element to perform the matching; in some cases, name matching may not be possible either. These are important practical considerations in the context of comparing and merging models [Alanen 2003] [Lin 2004].

A variety of graph algorithms can be used to manipulate the graph representations: aggregation or condensation algorithms can abstract a simpler graph from a detailed one [Choi 1990], [Cimitile 1995]; clustering algorithms can identify strongly connected components for re-modularization. Similarly, graph grammars [Baresi 2002], [Baresi 2005] are well suited for this general problem. In particular, triple graph grammars [Schürr 1995] can modify the source, target, and correspondence graphs simultaneously.
We can represent some of the previously discussed approaches as instances of this general problem. In [Rich 1990], a program can be represented as a source graph; a cliché library can contain a number of possible target graphs. Recognizing a cliché from the library involves finding a correspondence graph between the source graph and one cliché graph. In the Reflexion Models [Murphy 2001] approach, the source graph is the Source Level Model, the target graph is the High Level Model, and the correspondence graph is maintained with a set of regular expressions. In the case of object-oriented refactorings, both the source graph and the target graph are UML diagrams with additional semantics.

**Post-Research Maturity**

Re-engineering technologies are at different stages of maturity in the Redwine-Riddle model. More studies along the lines of [Sneed 1990] [Leach 1997] should be conducted to measure the effect of re-engineering upon software maintainability.

**The Scalability Challenge**

Scalability will remain a challenge for the foreseeable future. Many of the graph algorithms that are needed, such as sub-graph matching for detecting code duplication or finding design patterns in code, are NP-Complete. This requires using approximation algorithms if they exist. Even the simplest AST-based manipulations have to deal with the fact that an AST can be an order of magnitude larger than the corresponding source text [Bowdidge 1998]. Computing alias information can require between $O(n)$ and $O(n^3)$ time the number of variable references in the program depending on the algorithm [Griswold 1995]. There is promise as some startup companies are adopting many research techniques with scalability as the main consideration [Baxter 2004].

**Continuous Re-engineering**

The agile software development community and in particular, eXtreme Programming (XP) encourages a culture of continuous reengineering [Mens 2004]. The codification of re-engineering patterns, such as the ones in [Dewar 1999] and [Demeyer 2002] is an encouraging sign towards the external exploration of such techniques.

**Model-Driven Architecture (MDA)**

Concepts from transformational programming [Balzer 76] are being re-adopted today by OMG’s Model Driven Architecture (MDA) [OMG], which advocates the separation of the Platform-Independent Models (PIM) from the Platform-Specific Models (PSM). Model-to-model transformations where a transformation specification is used to generate a target model from a source model is the cornerstone of MDA. If MDA becomes a reality, re-engineering will simply consist of changing the model and performing code generation on the updated model.
Conclusion
In his keynote talk, [Brössler 2001] notes that software re-engineering projects are not very popular and that there is a wide gap between research, education, and practice. Academic and professional software engineering education still heavily emphasizes new languages and techniques targeted towards building the “Perfect System” with little regard to the evolution of such systems. [Rugaber 1996] notes that “re-engineering research has had notably little effect on actual software reengineering practice. Most of the published papers in the field present techniques supported by prototype tools; few of which have actually been used on real projects”. This statement may be overly pessimistic: some re-engineering techniques, such as refactoring, introduced in 1992 are at the beginning of their Popularization stage in the Redwine-Riddle maturation model; refactoring support is now available in freely available open-source Integrated Development Environments (IDEs) [Eclipse].

As indicated by the sheer size of today’s legacy systems, scalability is an important consideration and a major impediment for the industry adoption of many research ideas. In fact, one may argue that the techniques that are still in the internal exploration stage of the Redwine-Riddle model are precisely those that did not scale, e.g., the transformational implementation approach. Many techniques, such as the use of coarser-grained abstractions and hierarchy, are needed for scalability. The ability to automate a transformation is also critical, and may explain why some approaches, e.g., using category theory to compute changes [Wiels 1998] are still at the Basic Research stage in the Redwine-Riddle maturation model while other approaches are reaching the stage of external exploration. A tool solidly grounded in theory based on graph grammars, with a graphical representation based on UML, FUJABA, received the 2004 IBM Eclipse Innovation Award, and is available as open source software.

This survey attempted to cover a representative sample of the research literature on re-engineering, and does not claim to be exhaustive. In particular, this technology survey did not cover non-technical aspects of re-engineering. The interested reader is referred to the following papers as starting points on the economics of re-engineering [Sneed 1991]; studies of re-engineering project failures including non-technical reasons [Bergey 1999]; frameworks to identify high-value systems which would benefit from re-engineering [Bergey 1997] [Ransom 1998]. The Redwine-Riddle model clearly states that technologies do not fail or succeed in a vacuum: there are additional organizational or human inhibitors and facilitators that may affect their maturation rates.

References


[Compuware] Compuware OptimalAdvisor (was Package Structure Analysis Tool). http://javacentral.compuware.com/pasta/


