Architecture-based software reengineering

Isnet62 Project

Philippe Dugerdil

February 2006
1. INTRODUCTION ............................................................................................................. 5
  1.1 Reverse engineering and reengineering ................................................................. 5
  1.2 Reverse-engineering ............................................................................................... 6
  1.3 Reverse engineering legacy systems ....................................................................... 8
  1.4 The re-engineering life-cycle .................................................................................. 10

2. CURRENT TRENDS IN RE-ENGINEERING AND ARCHITECTURE RECOVERY .... 12
  2.1 Architecture-recovery systems ................................................................................ 12
    2.1.1 Rigi ................................................................. 12
    2.1.2 SAR (Krikhaar) .................................................... 14
    2.1.3 Discussion .......................................................... 17
  2.2 Slicing-based reengineering techniques .................................................................. 17
    2.2.1 Introduction to program Slicing ........................................................................ 17
    2.2.2 Using slicing techniques in reengineering .......................................................... 21
  2.3 Formal concept analysis-based reengineering systems ........................................ 25
    2.3.1 Brief introduction to the theory of Formal Concept Analysis ......................... 25
    2.3.2 Module Identification ....................................................................................... 26
    2.3.3 Module restructuring ....................................................................................... 28
    2.3.4 Linking features to source code ........................................................................ 29
    2.3.5 Discussion ......................................................................................................... 31
  2.4 Other trends ............................................................................................................ 31

3. GOAL OF THE PROJECT ............................................................................................... 34
  3.1 Introduction ............................................................................................................. 34
  3.2 Business driver ....................................................................................................... 35
  3.3 Quality driver .......................................................................................................... 36
    3.3.1 Quality attributes ............................................................................................. 36
    3.3.2 Architecting software for quality ....................................................................... 37
  3.4 Mixed driver ............................................................................................................ 37

4. PROGRAM COMPREHENSION AND REENGINEERING .................................... 38
  4.1 A simple model of program structure comprehension ......................................... 39
  4.2 Basic definitions ..................................................................................................... 39
    4.2.1 Program model, domain model ......................................................................... 39
    4.2.2 Discussion ......................................................................................................... 41
  4.3 Interpretation ........................................................................................................... 41
  4.4 Understanding Program Structures ......................................................................... 41
    4.4.1 Understanding programs .................................................................................. 41
    4.4.2 Discussion ......................................................................................................... 42
    4.4.3 Meaningful architecture .................................................................................. 42
    4.4.4 Understanding complex structures ................................................................. 42
    4.4.5 Discussion ......................................................................................................... 43
  4.5 Building a domain model and its interpretation .................................................... 44

5. BUSINESS MODELING AND REVERSE ENGINEERING .............................. 45
  5.1 Introduction ............................................................................................................. 45
  5.2 Size of the recovered components .......................................................................... 45
  5.3 Basic hypothesis ..................................................................................................... 46
  5.4 Architecture views to be recovered ....................................................................... 46
  5.5 Business process terminology ................................................................................ 47

6. THE REVERSE ENGINEERING PROCESS .................................................. 48
  6.1.1 Introduction ......................................................................................................... 48
| 6.1.2 | Overview of the process | 48 |
| 6.2 | Business scoping | 49 |
| 6.2.1 | Introduction | 49 |
| 6.2.2 | Workflow of the discipline | 50 |
| 6.2.3 | Specify the business scope | 50 |
| 6.2.4 | Specify the target quality attribute | 51 |
| 6.3 | Domain modeling | 51 |
| 6.3.1 | Workflow of the discipline | 51 |
| 6.3.2 | Redocument the business use case | 52 |
| 6.3.3 | Redocument the entity model | 52 |
| 6.3.4 | Build the data dictionary | 53 |
| 6.3.5 | Redocument the business analysis model | 53 |
| 6.3.6 | Redocument the system use-cases | 56 |
| 6.3.7 | Build the system analysis model | 57 |
| 6.3.8 | Build the feature cross reference | 60 |
| 6.3.9 | Working by iterations | 60 |
| 6.4 | Architecture recovery | 62 |
| 6.4.1 | Introduction | 62 |
| 6.4.2 | Workflow of the third discipline: Architecture recovery | 64 |
| 6.4.3 | Assess the current quality of the system | 65 |
| 6.4.4 | Redocument the visible high level structure of the system | 65 |
| 6.4.5 | Identify the target work task & corresponding use-cases | 66 |
| 6.4.6 | Run the use-cases and record execution traces | 66 |
| 6.4.7 | Map the traced functions to the visible high level structure of the code | 67 |
| 6.4.8 | Generate the call graph for the traced functions | 68 |
| 6.4.9 | Bottom-up validation of the analysis model | 70 |
| 6.4.10 | Rebuild a meaningful architecture | 74 |
| 6.4.11 | Assess the architecture | 83 |
| 6.4.12 | Make re-architecture proposal | 83 |
| 6.4.13 | Extract knowledge from the code | 83 |
| 7. | CONCLUSION | 84 |
| 8. | REFERENCES | 86 |
Abstract

For the last 15 years software reengineering has become an important field of computer science and an active field of research. In fact, the large usages of information system in enterprises, that automate ever more critical tasks, make the enterprise very dependent of their information systems. But these systems may have been built and maintained for many years leading to what is usually called “legacy systems”. Among the reasons to reengineer a system, the enhancements of its non-functional qualities play an important role (maintainability, performance, security, portability and the like). On the other hand, these qualities are very dependent on the architecture of the system. However, the architecture is rarely explicit in the documentation of a legacy system, if any documentation is available at all.

On the other hand, an important step in software reengineering is to understand the system before acting on it. In the absence of any documentation, the models of the software must be rebuilt. Among them, the model of the architecture plays an important role. It helps to lower the complexity of the software by grouping the software elements in clusters or components. In the literature, many techniques have been proposed to recover such a structural model. Consequently, the report starts with a summary of the main trends in software architecture recovery. Then we propose a small model of software understanding and discuss its role in software architecture recovery. From this preliminary work, we present a software architecture recovery process that is based on the modeling of the business processes supported by the software. This method, which is close to the Unified Process for software development, works both top down (from the domain concepts to the software artifacts) and bottom-up (from the software artifacts to the domain concepts).

How complex or simple a structure is depends critically upon the way we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve this simplification, we must find the right representation.

1. Introduction

1.1 Reverse engineering and reengineering

For the last 15 years software reengineering has become an important field of computer science and an active field of research. In fact, the large usage of information system in enterprises, that automate ever more critical tasks, make the enterprise very dependent of their information systems. But these systems may have been built and maintained for many years leading to what is usually called “legacy systems”. These systems, like any others are subject to what are called the “Lehman laws” of software maintenance:\footnote{A legacy Information System is any information system that significantly resists modification and evolution to meet new and constantly changing business requirements….To complicate matters, these IS are mission-critical- that is essential to the organization’s business - and must be operational at all times”. [Bro95].}

1. The law of continuing change: A program that is used in a real-world environment must change or become progressively less useful in that environment.

2. The law of increasing complexity: As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.

The first law comes from the fact that the technical, economical, legal, administrative and social environment of an enterprise change, leading to a corresponding need to modify the supporting information systems. The problem is that, generally, not enough resources are invested to counterbalance the effect of the second Lehman law. Then the system has a tendency to become more and more complex as time passes. Jacobson even suggests that the increase of complexity following a change to the system, or the increase of its “entropy”, is proportional to the entropy of the system before the change is made [Jac92]. In other words, the more complex the system before the change, the higher the increase of its complexity. Jacobson then uses this idea to suggest that an IT system whose complexity is not actively controlled reaches a point where its complete rewriting becomes compulsory. Due to the large amount of the software budget devoted to maintenance (from 60 to 90% of the total software budget depending on the authors [Erl00, Kosk]), it is clear that any initiative to facilitate maintenance could be very profitable.

Beside the increase of complexity, we generally observe that the modifications to a system are rarely reported in its documentation. Then, with time, the implementation of a system will not be aligned with its documentation. This discrepancy can be seen at the architectural level (the difference between the “as-designed” and the “as-implemented” architecture [Kaz97]) as well as the functional level (new user-level function without design documentation). In such a situation it is useless and even misleading to rely on the documentation to maintain a software system.

Whatever the outcome of the reengineering of a working legacy system\footnote{Cited in Rainer Koschke’s Thesis [Kos00] and by Ivar Jacobson in his book on use-case driven software development [Jac92]. These laws were originally expressed in [Bel76] in a somewhat different form.}, the engineer must know how the system works and what its actual business rules are\footnote{This reengineering could range from the mere restructuring of the code to its complete rewrite.}. In short, sufficient information must be extracted from the current legacy system for the reengineering goal to succeed. The quantity of information will depend on the reengineering task at hand.\footnote{Which may well differ from what has been initially documented [Nin93].}
For many reasons, especially economical, the redevelopment of information systems from scratch seems less of an option today [Gal99]. In the case of the re-engineering of a legacy system, one possibility is to reuse some of the legacy components, whose working is stable and proven, in a new system using a modern development technology such as object oriented environments.

In fact, the re-engineering of complex software systems is an active research topic ranging from the re-documentation of software architecture, to the understanding of programs, the reuse of software components or the restructuring/modernization of systems to extend their work life.

Before going any further, it is useful to define the main terms used in the reverse engineering literature. For this, we will follow the well known categorization of Chikofsky & Cross [Chi90]:

**Forward engineering**: this is the traditional way of designing systems, starting from abstract logical and implementation independent specification to gradually lead to the implementation of a physical system.

**Reverse engineering**: process under which an existing software system is analyzed to identify its components and the relation between them and to create representation of the system at different conceptual levels. In this area one generally identifies two sub-domains: re-documentation, where a representation at the same level of abstraction is created or enriched (such as the call graph from the source code) and design recovery where model of higher levels of abstraction are created. The latter is generally a prerequisite to the understanding of the program.

**Restructuring**: this is the transformation of a representation of a system to another one, at the same conceptual level, when keeping the external behavior of the system. Typically, one will restructure the source code of a system to enhance its readability.

**Reengineering**: analysis and modification of a software system to change its form and implement it in its new form. Generally, reengineering is made of two sub-steps: reverse-engineering to create an abstract description of the system, followed by forward engineering or restructuring.

Tightly related to the reverse-engineering activity is the domain of software architecture and reverse-architecting of software systems. In this report, we will use the definition of Bass et al. [Bas03] of software architecture:

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them.*

### 1.2 Reverse-engineering

Whatever the goal of the reengineering project, the first step is generally to understand the subject system, or at least its structure, with respect to its behavior. The outcome of this step may be used for various tasks ranging from architecture conformance evaluation (to check if the as-implemented architecture conforms to the as-designed architecture) [Kaz03], to legacy component identification and extraction [Ber01] or system restructuring and re-architecting. The basic framework of a reverse-engineering system is represented in the following figure (Figure 1).

For example, the first three steps represent the basic structure of well known software analysis systems such as the old DALI framework [Kaz97,Sea03] and its modern successor the ARMIN tool [Kaz03], the Rigi tool ([Mul93],[Won98]) and its extension “Shrimp” [Sto95],
or the CodeCrawler environment [Lan03]. They are close to Krikhaar’s presentation of the reverse-engineering process [Kri99] where the steps are called: “extract”, “abstract”, “present”. Interestingly, M.-A. Storey recently proposed a classification of software comprehension tools in 3 categories: extraction, analysis and presentation [Sto05]. In fact, the domain of software comprehension and re-engineering are closely related.

![Basic reverse-engineering framework](image)

The first step is to transform the system’s artifacts (primarily the source code, but also the file structure, directory structure, build and make files) to some abstract representation from which further manipulation could be performed. The result of this computation is stored in some kind of permanent repository. This step has two main advantages

- To facilitate the subsequent processing of the different information sources which are transformed into one single formalism;
- To eliminate the peculiarities of the source’s programming language.

However this is true only to a certain extent. In fact, when the programming paradigm changes (generally from a procedural style to an object-oriented one) then, to some extent, it must be reflected in the abstract representation (unless the parser/analyzer makes some paradigm transformation). It must be noted that the abstract representation is basically at the same level of abstraction as the original artifact, although some information could have been filtered out (noise reduction). Among the abstract representations one finds all graph-related data structures, partition relation algebra [Kri99], or proprietary formalisms like the famous Rigi Standard Format (RSF): the input format to the well known Rigi reverse-engineering tool ([Mul93], [Won98]) or the FAMIX format of the FAMOOS ESPRIT Project [Dem99].

The second step (view builder) “processes” the information from the abstract representation repository to build the “views” of the system. These views represent some higher-level abstraction of the information. Usually, a single view does not provide enough clues to understand the system.

The third step, the views processing, builds some meaningful human understandable abstracted view of the system by aggregating the information from multiple basic views. For example, the third step may abstract some higher representation of the architecture of the system or alternatively compute software metrics over the system. The result is often displayed as some form of graph structure, but not always. As an example, the CodeCrawler system [Lan03] builds a graphical representation that blends structural information and metrics.
1.3 Reverse engineering legacy systems

Given a legacy system to reengineer, different information sources could be tapped to reverse-engineer it, among which we find:

- The technical documentation of the system (specification, analysis, design, implementation, test, deployment documentation);
- The manuals: the user and training manuals;
- The expertise of the people involved in the development of the system and its maintenance: software architect, designer, developer and maintainer;
- The deployment artifact of the system itself: source code with comments, directory and file structure, deployment descriptors, build and compilation scripts;
- The other stakeholders of the system: the users and business experts.

According to Kruchten's ideas [Kru95], a software architecture may be represented through 4+1 views, which give 4 different "perspectives" on a software system⁵ (the fifth is the "use-case" view that influences the 4 other views. Then, in the reverse-engineering context, a source of information could be characterized by 4 attributes:

- abstraction level;
- scope ⁶;
- truthfulness ;
- associated views.

As the views of a system are correlated, missing or uncertain information in some view might be reconstructed or strengthened using the information of another view. In fact every source of information may only describe some limited part of the system and/or not be up to date. Moreover, the attributes of a "human" source of information may not be known to the source itself (for example, a developer may not know that the system has evolved since its initial development).

![Figure 2](attachment:image.png)

**Figure 2** Basic reverse-engineering framework augmented with other information sources

---

⁵ The 4+1 view of software architecture is now integrated in the popular Rational Unified Process (RUP) for software development [Kru00]. The idea of software views is not unique to Kruchten’s work. For example it is also present in the work of Hofmeister et al [Hof00]. Although not similar, the views of Hofmeister are close to those of Kruchten.

⁶ The degree of coverage of the system by the information source.
However, depending on the reengineering task at hand, the type of information to retrieve might be different. For example, if the goal of the reengineering task is to improve the architecture of a system, then we might not need to understand the smallest details of the computation algorithms. On the other hand, if the goal is to extract the business rules, then the computation algorithms might have to be investigated but not the overall architecture of the system.

According to their goal, the various reengineering frameworks and methodologies will favor some of the information sources to the detriment of others. Depending on the information source available, a reengineering process will need to infer the missing information to reach its goals. In fact, according to Kazman & Carrière, the reverse engineering process amounts to the work of a detective: to reconstruct a complete picture of the system from available evidence \([\text{Kaz97}]\). Hence, reengineering is tightly connected to the field of program understanding \(7\) which is coupled to the notion of program complexity \(8\). For example Banker et al. \([\text{Ban93}]\) have investigated the link between maintenance costs and software complexity (interpreted as the difficulty in the understanding of a program) and shown a positive correlation. In the past, many reengineering projects have failed, one of the reasons being the lack of awareness of the legacy system architecture \([\text{Ber99}]\). Hence, we strongly believe that a good understanding of the system’s architecture is key to the success of any substantial reengineering project \(10\). In this report, according to \([\text{Chi90}]\), reengineering will be considered as two complementary activities: reverse engineering and forward engineering.

Due to the gaps in the information sources, one should augment the information at each step of the framework by tapping other information sources (documents, people…) \(\text{(Figure 2). As the information might not be totally truthful, some correlation between sources will be necessary.}\)

The goal of our research work is not to find a way to completely describe the architecture of a given legacy software system, but to get enough understanding so that its reengineering goal has some chances to be reached. Then, the architectural analysis will always be driven by the reengineering project at hand, which itself will be driven by the business needs.

Although the business need for the reengineering of a software system is often explained in the literature, very few papers make an actual use of the business context and business goal of the project in particular the business processes supported by the software. This is the starting point of our own research.
1.4 The re-engineering life-cycle

The reverse engineering of a system could seek many objectives among which we find:

- Understand the current system so that maintenance is easier and less error prone;
- Reverse-document the specification of the system so that they closely match the running system. This could then be used to develop or subcontract the development of a new system developed from scratch or even configure an ERP.
- Re-architect the running system to improve its quality attributes.
- Extract useful component to be reused in a new system.

In the case of software reengineering based on architecture, the SEI institute published the famous Horseshoe model [Ber99b]:

![Horseshoe model for legacy system reengineering](image)

In this model, the reengineering process is made of three fundamental steps:

1. Analysis of the existing system and building of a set of logical description of its structure.
2. Transformation of these descriptions to get some new, improved structure.
3. Restructuring of the system to follow these new descriptions.

In this model, the source code is analyzed and models of increasing level of abstraction are created up to the architectural level. Then the system is re-architected to match the new specifications for the system or to reach a predefined level of quality attribute. Since the quality attributes are driven by the system architecture, it is first necessary to recover this architecture to propose any enhancements. The round trip through all the levels of the horseshoe represents the most complete form of reengineering. However, in practice, there are two short paths through the horseshoe. They are represented by the transformations at the two lower levels of abstraction: the code level and the functional level. At the code level, these transformations can range from simple actions such as syntactic replacements in the source code to the porting of a system from one programming language to another one. The changes in the implementation of a function, the adaptation of a function to new requirements or the adaptation of the interface of a function are examples of transformations.
at the functional level. Architectural level transformations are represented, for example, by changes to the structure of the system, the type of components and interactions or the allocation of function into modules. Although lower-level transformations can take place without higher level transformations, higher level transformation are supported by lower level transformations [Ber99b]. For example, if one moves a monolithic program to SOA\(^{11}\), this operation is architectural because it implies changes to the very structure of the system. Such a fundamental change will have an impact at the functional and code levels too. Finally, it is worth noting that the higher the level of transformation to perform the higher the required level of human expertise.

The left arrow of the horseshoe model represents the reverse-engineering of the subject system. After the architectural transformation, the right arrow represents the forward engineering (restructuring) of the new system based on an improved architecture. This is why the SEI designed a new system development methodology called the Architecture-Based Design Method [Bac00].

\(^{11}\) Service Oriented Architecture
2. Current trends in re-engineering and architecture recovery

2.1 Architecture-recovery systems

2.1.1 Rigi

One of the most famous software architecture and design recovery systems is that of Müller et al. around the Rigi project [Mul93, Won98]. Rigi is basically a programmable software analysis and clustering engine coupled with a graphical editor called Rigiedit. The programming language for Rigi is TCL (Tool Command Language) that let users to define their own clustering procedures for example. The input to Rigi is the description of a graph of software element in a standard format called RSF (Rigi Standard Format). The translation from any input source code format to RSF is left to the user of Rigi, although the system provides some parser for C and Cobol. This is the first step of a reengineering framework as represented in Figure 1.

Since RSF has been the chosen abstract representation in several other reengineering frameworks, it deserves some deeper presentation.

RSF is a sequence of triples \(<\text{verb, subject, object}>\) that represent the source code structure as a graph of nodes and arcs. The elements of these triples are specific to a given domain i.e. the programming language and environment used. A domain model, that determines what node and arc types are possible together with their attributes, must be communicated to the Rigi system. This is done through a set of 5 files:

1. \textit{riginode} : lists the names of the types of the nodes. Each name is listed on a different line.
2. \textit{rigiarc} : lists the types of the edges. Each line holds the following kind of declaration : \(<\text{arcTypeName}> [<\text{startNodeTypeName}> <\text{endNodeTypeName}>]>. \textit{rigiattr} : defines the attributes of the nodes and arcs. Each line holds the following kind of declaration :
   - \text{attr <nodeName> < node attribute name>}
   - \text{attr <arcName> < arc attribute name>}
3. \textit{rigicolor} : declares the color of the nodes and arcs.
4. \textit{rigicolor} : declares a TCL script to be run when switching to this domain.

Given a software system to analyze, the Rigi user would first build a parser / analyzer that will produces the RSF format according to some domain model. Then he will create the 5 files to communicate this model to Rigi.

Once loaded into Rigi, the RSF representation of the software system is manipulated through the Rigi primitives, or user-programmed functions, to filter information and cluster elements into subsystems. However, the clustering is not automatic. In fact, Muller highlight that “Discovering subsystem structure is an art”. His approach “…is based on the premise that, given sufficient time, an experienced software engineer is usually able to decompose a system better than an automatic procedure can”. This means that the software engineer will inject some of his knowledge in the “view builder” step of our framework. This process is highly interactive: the user applies some filter function to simplify the displayed graph of software elements and then selects a set of elements to be grouped by Rigi. Then, Rigi will collapse the selected set to a single node, to which all the arcs coming from the outside of the new node will be connected. This could represent a candidate component or subsystem.
Later, the Rigi user can use zooming techniques to display the inner structure of the components.

Once some clustering is done, the user can invoke quality measurement primitives based on the notion of exact interfaces. These measures will quantify the level of encapsulation and separation of concerns of candidate components (or internal cohesion and external coupling) [Mul93]. One very interesting point is the set of techniques used to manipulate and simplify the graph [Mul90, Mul93]. We believe these techniques to be useful in many reengineering system. They are listed below.

- **Remove omnipresent node**: this let the user to reduce the noise in early stages of the clustering process. The idea is that a node with very high number of incoming arcs might represent a utility procedure (error logging, printing, debugging...). Then, the user could remove all the nodes whose number of incoming arcs is greater that a given threshold to simplify the graph.

- **Measure of the interconnection strength** $IS(n_1, n_2)$: compute the weight of the interconnection between any two nodes $n_1$ and $n_2$, which is proportional to the syntactic objects “exchanged” between the nodes. For example, in an OO system, one takes into account the types of the parameters of the messages between the objects. Next one defines two threshold: $Ts$ (strongly coupled), and $Tl$ (loosely coupled) with $Th > Tl$. If $IS(n_1, n_2) >= Th$, the two nodes will be considered as strongly coupled and collapsed into the same subsystem. On the other hand if $IS(n_1, n_2) <= Tl$, the two nodes will be considered as loosely coupled and collapsed into separate subsystems. If $Tl < IS(n_1, n_2) < Th$ no decision can be taken.

- **Compose by common neighbor**: Let $CS(n_1, n_2)$ be the set of common “clients” of $n_1$ and $n_2$. A node $x$ is “client” of a node $y$ if there is some resource flow from $y$ to $x$. Let $SS(n_1, n_2)$ be the set of common suppliers of $n_1$ and $n_2$. A node $x$ is supplier of a node $y$ if there is some resource flow from $x$ to $y$. Then if $CS(n_1, n_2) > Tc$ or $SS(n_1, n_2) > Ts$, for some threshold $Tc$ and $Ts$, $n_1$ and $n_2$ are said to be common neighbors and can be collapsed into the same subsystem.

- **Compose by centricity**: let the external strength of a node $n$, $ES(n)$, be the sum of the weights of all the edges between $n$ and all other nodes that are not subsumed by $n$ (not collapsed into $n$) in the graph. Then, a node $n$ is said to be central if $ES(n) >= Tk$, and is said to fringe if $ES(n) <= Tf$, where $Tf < Tk$ are some predefined thresholds. Normally fringe and central components should not be grouped into the same subsystem. On the other hand, the change to a central component may affect a lot of other components and must therefore be performed with care. Moreover, a node $n$ for which $ES(n) = 0$ represents some dead code (no clients nor suppliers) and can be filtered out. This technique is a kind of noise reduction.

- **Compose by name**: this is the simplest grouping technique which is based on the nodes names: we group the nodes with similar names using some pattern matching engine. Although this technique could be useful in the case the designers of the system have stuck to some standard naming convention, it could also identify unrelated components due to accidental matches.

To a certain extent, an omnipresent node could represent some aspect[12] of the software that is scattered over a set of modules, for example some logging functions. However, Rigi does not explicitly deal with the notion of aspect. While some of the graph simplification heuristics of Rigi are of widespread use in reverse-engineering environments, others are more specific to Rigi like Remove omnipresent node or Compose by centricity.

---

2.1.2 SAR (Krikhaar)

Although the work of Krikhaar [Kri99] is a bit old now, it is interesting in the sense that the abstract representation of the software artifacts are not based on some graph representation formalism but on the more formal language of algebra. In fact SAR (Software Architecture Reconstruction) is based on the Relation Partition Algebra (RPA). Since this is one of the few frameworks that do not rest on graph representation, we will present it in some details.

The approach of Krikhaar consists of a set of documented information retrieval and/or reconstruction steps to rebuild the software architecture. Each step shows how to tap the available information sources to get the information that is needed to perform the step. In fact, SAR takes explicitly into account any source of information and documents it in a common format. The information retrieval/reconstruction steps are separated in two categories: the InfoPacks and the ArchiSpecs. The InfoPack is “a package of particular information extracted from the source code, design documentation any other source”. The ArchiSpec is “a view on the architectural structure that makes explicit a certain architectural structure”. Both categories are documented in a common format:

- Name
- Context: architectural view\(^{13}\) to which the information belongs and identification of the related steps.
- Description: explains the contents of the step
- Method: description of the activities that must be performed to build the step.
- Example: real world example in which the method was applied.
- Discussion: supplementary information on the technique.

The InfoPacks and ArchiSpecs are structured as a directed graph representing the dependencies between the information retrieval/reconstruction steps: what step is required to perform another one. It is important to highlight that SAR makes an explicit use of all the source code information available such as the directory structure, the file structure, the compilation, build and import declarations or any other aspects of the software.

Moreover, the ultimate SAR goal is to improve the architecture of a system. Then, in a way similar to the SEI/CMM 5 levels [Hum89], Krikhaar defines 5 levels of software architecture reconstruction:

1. Initial level: starting level, where the architecture of the system is not known nor documented
2. Described level: the current architecture of the system is re-documented
3. Redefined level: this is the level where the architecture has been improved.
4. Managed level: at this level, the architecture is monitored to avoid its degradation (following the Lehman laws)
5. Optimized level: the architecture, which is under control, is optimized for future extensions.

Consequently the InfoPacks and ArchiSpecs are grouped according to the reconstruction level at which they work.

The heart of each InfoPack and ArchiSpec is the method part i.e. the way the information is retrieved/reconstructed. When possible, it is represented as a set of algebraic relations that could be used to build or query the software architecture model. The technique rests on some primitive relations that must be created by hand from the observation of the source code artifacts. Let’s now look at an example application of Archispecs and InfoPacks.

\(^{13}\) In the sense of Kruchten [Kru95].
The first step to understand a software system from its source code is to clarify the software concept used in the description of the system (i.e. the notion of component, subsystem, package, archive, file, function, type...). This is the purpose of the Software Concept Model ArchiSpec. It must be constructed iteratively by hand with the help of the available documentation and some discussion with the architect of the software system. The result can be represented as a UML class diagram with aggregation and composition associations as well as other relations.

Figure 4  Example of a Software Concept Model in SAR

However, this could be very difficult to do when faced with an unknown system. If the developer/architect of the system cannot be reached and no reliable documentation is available we may well not retrieve the software concepts the developer had in mind, beyond the file level.

Now comes what we believe to be the most difficult Archispec when faced with an unknown system: the Source Code Organization. It is intended to reconstruct three kinds of information:

1. The description of the source code organization (archives, versions, directories, files);
2. The mapping of these source organization elements into software concepts;
3. The process of retrieving the files from the software management system.

Although the first could be relatively easy to retrieve, the second one is at the core of most of the difficulties in reverse-engineering systems. In fact, as acknowledged by Krikhaar in his discussion of this Archispec: “The top-level concepts (e.g. subsystems) in the Software Concepts Model are often not reflected on any tangible item of the Source Code Organization. There is a risk of the meaning these concepts, which exist only at a more conceptual level, degenerating with time”. This is an essential problem in reverse-engineering: how to group software elements into meaningful software concepts (which may later be mapped to domain concepts). We will discuss this problem at length in the presentation of our own methodology.

From the Source Code Organization ArchiSpec we can retrieve the set of all the files that pertain to a given version of the system. The latter is represented by the File InfoPack. Then, by inspecting each of these source files, we may retrieve the dependencies between the files declared by an import statement. This is one of the roles of the Import InfoPack. Its result can be represented by the import Files, Files relation.

Starting from the results of the Software Concept Model and Source Code Organization Archispecs, a further step is to retrieve the containment relationships between the actual software elements (what function belongs to what component, what component belongs to what subsystem and so on). This is the role of the Part-Of InfoPack. Depending on the programming language used, we could group statements into functions, functions into classes or files, files into packages. But the next level of software concept is generally much more difficult to retrieve. Although some of the information could be inferred from the inspection of the directory structure of the code, we must often exploit other knowledge sources such as the software architect of the system. Again, to retrieve this information can be very difficult depending on the availability of a reliable knowledge source to supplement the raw information provided by the source code. The result of this InfoPack can be represented by the partof Files, Pak, partof Pack, Comps and partof Comps, Subs relations, where
Packs is the set of all the packages, Comps is the set of all components and Subs is the set of all subsystems.

The information retrieved by the previous Archispec and Infopacks can be used to infer the component and subsystem relationships using the RPA. This is the purpose of the Component Dependency Archispec, which uses a chain of lift operations:

\[
\text{imports}_{\text{Packs};\text{Packs}} = \text{imports}_{\text{Files};\text{Files}} \uparrow \text{partof}_{\text{Files};\text{Packs}} \\
\text{imports}_{\text{Comps};\text{Comps}} = \text{imports}_{\text{Packs};\text{Packs}} \uparrow \text{partof}_{\text{Packs};\text{Comps}} \\
\text{imports}_{\text{Subs};\text{Subs}} = \text{imports}_{\text{Comps};\text{Comps}} \uparrow \text{partof}_{\text{Comps};\text{Subs}}
\]

Where the new relation \( R \uparrow P \) built from a relation \( R \) and a part-of relation \( P \) using the lift operator \( \uparrow \) which is defined as: \( R \uparrow P = \{ <x; y> | \exists a; b <a; b> \in R \land <a; x> \in P \land <b; y> \in P \} \). In this case \( x \) and \( y \) are names of sets and \( <z; s> \in P \) is true if \( z \) is an element of the set named \( s \).

The computation of this import relation shows how RPA can be used to actually compute abstract, higher level, information from the knowledge of elementary relations. This highlights the power of this approach. But, again, the latter rests on the discovery of the fundamental relations between the software concepts and source code elements as well as the containment relation between software elements which is very difficult. The work of Krikhaar does not give many clues as to the way to retrieve this information in the absence of a reliable external knowledge source. But, once this information is retrieved, SAR could be used to infer higher level information. This technique is similar to the idea of view fusion of DALI [Kaz98b], where the RPA inferences are replaced by SQL queries.

As for the redefined level of software architecture, Krikhaar advocates the use of the software model built during the reconstruction phase to test restructuring ideas based on architectural simulations. These simulations rest on the architectural description represented in RPA. Here are the Archispec that help to transform the architecture:

Component Coupling quantifies the coupling between dependent components. This dependency is computed by the Component Dependency Archispec. For example, we can use metrics like the number of relation between components at the lower level, the number of entities used by a component, the number of other components using a given component. This technique is similar to the Rigi’s interconnection strength metric between two nodes \( IS(n_1,n_2) \) when the nodes are components. It must be noted however, that the lowest granularity of the entities to be shared by components in SAR is the file level. This is much too coarse in many situations. For example, the number of function called between components plays an important role in the coupling strength between components. Moreover there are more subtle dependencies like variable sharing that are not taken into account in SAR. Although not impossible in theory, the use of the SAR approach at the finest granularity level like functions and variables may well lead to an intractable complexity.

Cohesion and Coupling Archispec quantifies the intra-connectivity and inter-connectivity of components. Cohesion (intra-connectivity) of a component is defined as the ratio between the actual dependencies of its parts to the number of all possible dependencies. Coupling (inter-connectivity) of a component is defined as the ratio between the actual dependencies between the parts of two components to the number of all possible dependencies. Again, this is close to the Rigi’s metrics.

Aspect Assignment Infopack explicitly address the aspects of software like logging, test, diagnostic, development, installation and so on. Let Asps be the set of all aspects present in
the software. The main result of this Infopack is represented by the relation addresses Files, Asps. The first step of the Infopack is to determine all aspects of the software. This could be easy if the aspects have explicitly been dealt with during design. But it is much more difficult if this is not the case. One could again try to exploit other available knowledge sources such as the designers and architects of the software. The second step is to locate the aspects into the software. Sometimes, this can be done by analyzing the names of the files or packages if the aspects have been handled explicitly. Krikhaar makes the following statement: “To reconstruct Aspect Assignment, we should take the decomposition level that fits such an assignment best”…”If aspects appear only at statement level, it is practically impossible to obtain a useful addresses relation”.

Aspect Coupling Infopack starts from the idea that the software elements that implement the aspects are well separated. A single file would then address exactly one aspect. For example, the file that implements some aspect $a_1$ is separated from the file implementing $a_2$. Given the addresses Files, Asps and imports Files;Files relations, it is possible to derive the depends Asps, Asps relation that shows what aspects depend on what other aspect.

2.1.3 Discussion

Although this idea of Archispec-based restructuring simulation is appealing, we must recall that the software model in RPA does not represent the functionality of the software but only its structure. Hence we do not clearly see how we could decide on a system restructuring, based only on the proposed Archispec without a proper understanding of the system. Krikhaar seems to draw the same conclusion when he writes, in one of the example he gives of the Cohesion and Coupling Archispec: “This discussion shows that a proper understanding of the system is required to be able to draw proper conclusions from metrics” ([Kri99], p110). When analyzed in the light of the notion of software quality attributes [Bas03], SAR concentrates on the maintainability of the code. Then the other software quality attributes such as performance, portability, testability, and usability are not addressed. Even if the SAR method might seems difficult to apply on a real reengineering situation when proper knowledge sources are unavailable (i.e. the architects and designers of the system and some up to date documentation), it does a very good job at identifying the pieces of information needed for such a work. It shows step by step what information should be retrieved to redocument and restructure the architecture of the system.

2.2 Slicing-based reengineering techniques

2.2.1 Introduction to program Slicing

Program slicing techniques have long been used in program analysis, maintenance and reengineering and originate from the work of Mark Weiser [Wei81]. Today, it is used either alone [Ver03] or in combination with other techniques [Ton03] to help restructure and/or find components in a legacy source code. Slicing is a program decomposition technique whose goal is to find, in a source code, the set of statements that influences the value of a variable at some location. Informally, a slice of the program answers the following question: “what program statements potentially affect the value of the variable $v$ at statement $s$?”

The basic technique to compute a program slice can be decomposed in three steps. First we must transform the program to a control flow graph [Wei81, Bin96]. Then we compute the set of relevant variables for each node of the graph (the variables whose values could potentially affect the value of the target variable). Third we build the slice as the set of statement that assign a value to a variable in the set of relevant variables of each node.
A control flow graph (CFG) is a oriented graph \((S,D)\) where the nodes is the set \(S\) are the statements of the program and \(D\) is the control-dependency relationship. A node \(s_1\) is related to a node \(s_2\) and an edge will be drawn from \(s_1\) to \(s_2\), if \(s_2\) could get the control from (i.e. be executed immediately after) \(s_1\). This happen if [Aho86] :

- \(s_1\) is a branching instruction that transfers the control to \(s_2\)
- \(s_2\) follows immediately \(s_1\) and \(s_1\) is not a unconditional branching instruction.

Then \(s_1\) is called a **predecessor** of \(s_2\) and \(s_2\) is a **successor** of \(s_1\), the following figure gives a representation of such a CFG.

At each node \(n\) of the CFG we associate two sets, \(REF(n)\) to hold all the variables that are referenced at node \(n\) and \(DEF(n)\) to hold all the variables that are defined (assigned) at \(n\). For example in the figure above (Figure 5) the two sets are:

<table>
<thead>
<tr>
<th>Node</th>
<th>(REF(n))</th>
<th>(DEF(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a = 1;)</td>
<td>(a)</td>
</tr>
<tr>
<td>2</td>
<td>if (a &gt; 0) then</td>
<td>(a)</td>
</tr>
<tr>
<td>3</td>
<td>(b = 1;)</td>
<td>(b)</td>
</tr>
<tr>
<td>4</td>
<td>(a = a + b;)</td>
<td>(a,b)</td>
</tr>
<tr>
<td>4</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(a = a - b;)</td>
<td>(a,b)</td>
</tr>
<tr>
<td>6</td>
<td>endif;</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>(\text{print}(a))</td>
<td>(a)</td>
</tr>
</tbody>
</table>

Figure 5  A sample program with its control flow graph

Figure 6  Def() and Ref() sets for the nodes of the sample program

**Definitions**

A **slicing criterion** is a pair \(<s,v>\) where \(s\) is a statement and \(v\) a variable referenced in this statement.

A **slice** of a program \(P\) for the slicing criterion \(<s,v>\) is the set of statement of \(P\) that capture the behavior of \(v\) at \(s\) (in fact just before the statement \(s\) is executed).

Intuitively the statements in the slice are those that, when executed, influence the value of \(v\) at \(s\). However, depending on the type of slicing considered, the notion of “capture” can be a
little bit different from this intuitive meaning. Then another definition of a program slice rests on program execution. In this view, a slice of a program P for the criterion <s,v> is a reduced program (a program obtained by deleting some statements from P) which compute the same sequence of value of v at s when inputting the same value in both programs. In summary, depending on the author, the definition may imply that the slice be executable or not. Then, this may lead to different slices for the same slicing criterion. Usually, these techniques are called static slicing because they involve an exhaustive static analysis of the source code of the program.

The computation of a slice S with respect to the slicing criterion <s,v> is basically done through the following stages:

1. Compute the set relevant(n) of relevant variables for each node n of the CFG of the program. This represents the dataflow of the program.
2. Use this dataflow to compute the slice with the following two steps:
   a. Add to S each node n that assign a variable of the set of relevant variable of m, where m is an immediate successor of n in the CFG.
   b. Add to S each node of the slice relative to the new slicing criterion <k, ref(k)> where k is a control statement that directly controls the execution of some statement in S.

The set of relevant variables at a node n represent the variables whose values, at the time of the execution of n, could potentially affect the value of v at s.

The computation of the set of relevant variables for the slice relative to <s,v> works as follows.

1. Define the set of relevant variables at node s as the singleton {v} : relevant(s) = {v}.
2. For each non conditional statement node n, relevant(n) is defined with respect to the relevant set of m, where m is the immediate successor of n as:
   relevant(n) = (relevant(m) - DEF(m)) ∪ (REF(m) if relevant(m) ∩ DEF(m) ≠ ∅ )
3. For each conditional statement node n, relevant(n) is defined with respect to the relevant sets of all its immediate successors nodes m1,m2,… as
   relevant(n) = \bigcup_{m_i \in \text{control}(n)} relevant(m_i)
4. Work backwards from s repeating the steps 2 or 3 until the beginning of the program is reached from all branches of the CFG.

For structured programs, the computation of set of statements of the slice relative to <s,v> can now be computed as follows.

1. For each node n of the CFG, compute the set of conditional statements control(n) that directly controls the execution of n. Control(n) is either the empty set or a singleton set.
2. Add to the slice each node n for which def(n) ∩ relevant(m) ≠ ∅ , where m is the immediate successor of n.
3. Whenever a statement n is added to the set, add to the slice all the statements that represent the slice relative to <k, REF(k)> where k is the conditional statement that directly controls n: k ∈ control(n).

---

14 If the conditional statement has some side effect then, after the computation of union set, the resulting set is updated with the result of the computation of step (2) above for the side effect part of the statement.

15 In the case of an unstructured program, this set may contain several elements. But unstructured program brings some more problem as we will see later in the text.
Here we show an example of application of the slicing algorithm on a straight line program, i.e. a program without control statement. The computation of the slice relative to \( <7,a> \) is given by:

\[
\text{Program} \quad \text{DEF(n)} \quad \text{REF(n)} \quad \text{Relevant(n)} \quad \text{def(n) } \cap \text{relevant(m) } \neq \emptyset
\]

| Program   | DEF(n) | REF(n) | Relevant(n) | Control(n) | def(n) \( \cap \) relevant(m) \( \neq \emptyset 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>b = 2;</td>
<td>b</td>
<td>a</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>c = 3;</td>
<td>c</td>
<td>a</td>
<td>true</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>a = c + a;</td>
<td>a</td>
<td>c,a</td>
<td>c,a</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>b = c - 3;</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>c = 4</td>
<td>c</td>
<td>a</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>print(a)</td>
<td></td>
<td>a</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 7  Sliding a straight line program

Then the slice relative to \(<7,a>\) is the set of statements 1, 3 and 4.

Below, we show an example of the application of the slicing algorithm on a program with control statements. In this case, a new column that represents \( \text{control(n)} \) must be added to the table above. Below, we first represent the computation of the intermediary sets for the slice relative to \(<13,a>\), without including the slice computed for the control statement.

\[
\text{Program} \quad \text{DEF(n)} \quad \text{REF(n)} \quad \text{Relevant(n)} \quad \text{Control(n)} \quad \text{def(n) } \cap \text{relevant(m) } \neq \emptyset
\]

| Program   | DEF(n) | REF(n) | Relevant(n) | Control(n) | def(n) \( \cap \) relevant(m) \( \neq \emptyset 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>b = 2;</td>
<td>b</td>
<td>a</td>
<td>true</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>c = 3;</td>
<td>c</td>
<td>a</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>d = 2;</td>
<td>d</td>
<td>a,b,c</td>
<td>false</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>if(d &gt; 0) then</td>
<td>d</td>
<td>a,b,c</td>
<td>false</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>a = c + b;</td>
<td>a</td>
<td>b,c</td>
<td>b,c</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>b = c - 3;</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>else</td>
<td></td>
<td>a</td>
<td>8</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>c = 4</td>
<td>c</td>
<td>a</td>
<td>8</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>b = c + a;</td>
<td>b</td>
<td>a,c</td>
<td>a,c</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>a = b – 7</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>endif</td>
<td>a</td>
<td></td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>print(a)</td>
<td></td>
<td>a</td>
<td>false</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 8  Slicing a program with control statements, intermediary slice

Without taking the control statements into account, the intermediary slice relative to \(<13,a>\) is the set of statements: \( \{1,2,3,6,9,10,11\} \). Now we must add to it the slice associated to the control statement \(<5,d>\) (the else part of the conditional is also controlled by node (5)):

\[
\text{Program} \quad \text{DEF(n)} \quad \text{REF(n)} \quad \text{Relevant(n)} \quad \text{Control(n)} \quad \text{def(n) } \cap \text{relevant(m) } \neq \emptyset
\]

| Program   | DEF(n) | REF(n) | Relevant(n) | Control(n) | def(n) \( \cap \) relevant(m) \( \neq \emptyset 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>a</td>
<td></td>
<td></td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>b = 2;</td>
<td>b</td>
<td></td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>c = 3;</td>
<td>c</td>
<td></td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>d = 2;</td>
<td>d</td>
<td></td>
<td>true</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>if(d &gt; 0) then</td>
<td>d</td>
<td>d</td>
<td>false</td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

Figure 9  Slicing a program with control statements, slice associated to the control statement

The slice associated to the control statement i.e. with criterion \(<5,d>\), is \( \{4\} \).
Finally, the slice relative to \(<13,a>\) is: \(\{1,2,3,6,9,10,11\} \cup \{4\} = \{1,2,3,4,6,9,10,11\}\).

The slicing algorithms can compute the slice of a program without loops in a single pass. For programs with loops, the algorithm must iterate over each loop until the set of relevant variables and the slice stabilize. It has been shown that the maximum number of iterations is the same as the number of assignments statements in the loop [Bin96].

In the case of unstructured programs, several solutions have been proposed in the literature. One is simply to transform the unstructured program to a structured one. However the problem is that the architecture of the program is changed which is not desirable in a reengineering work. In the case of programs using arrays, the situation is more difficult. In fact, the index of the array may be computed using arbitrary complex expressions. Then, in the general case, we cannot decide if two indexes are the same. This can only be done in some specific situations. One easy solution to this problem is to consider the array as a whole i.e. each time some array position is referenced in a node \(n\) then the whole array become part of the \(\text{DEF}(n)\) and \(\text{REF}(n)\). In reality, when assigning to value to some array position, the rest of the array stay unchanged. Then, globally, the new value of the array also depends on its old value. This is why the array becomes part of \(\text{REF}(n)\). This technique, although correct, leads to larger program slice that necessary. Records are much simpler to deal with. In fact we can work as if each of the fields was an independent variable. Pointers are another source of hard problems since we must compute all the possible locations a pointer could point to. The associated techniques go beyond the scope of this report. Slicing program across procedure calls raise yet another set of problems since we must deal with parameter passing between procedures. See [Bin96] for a deeper discussion of all these topics.

On the other hand, some authors have proposed the use of Program Dependence Graphs (PDG) to compute program slices. PDG is a representation borrowed from the domain of parallelization of algorithms which explicitly represents the data and control dependencies in a program. The advantage is that, once the dependence graph is constructed, the slicing algorithm is very efficient.

Finally, dynamic slicing applies the same idea but to the execution of a program. A dynamic slice contains all the statements that affect the value of a variable at some program point for a particular execution of the program. Rather than working on the source code of the program the dynamic slicing algorithm work on a particular execution trace. The slicing criterion is defined with respect to a given variable at a given occurrence of some statement (in case of loops, the same statement may be executed several times). Usually, such a dynamic slice contains much less statements than a static slice using the same criterion. Moreover, a variant of this is to perform the slicing technique on a program where some of its input variables are fixed while the other input variables are unconstrained. This is called quasi-static slicing [Del01]. If all the input variables are unconstrained, we get a static slice. If all the variables are fixed, we get a dynamic slice.

### 2.2.2 Using slicing techniques in reengineering

One of the first variations on the Weiser’s slicing technique has been proposed by Gallagher and Lyle [Gal91] for the maintenance of programs. The central idea is to isolate, in a program, all the statements that influence the value of a given variable whose computation must be modified. Then, they designed a technique called decomposition slice to extract these statements with respect to a variable \(v\) (without referring to any statement). But the computation of a decomposition slice does itself use the traditional slicing algorithm. Using
this technique, the maintainer knows that all the impact of the change of the computation of \( v \) will be limited to the slice.

Beck and Eichman [Bec93] highlight the benefits of program slicing techniques for reengineering since they allow to “reabstract” an implementation even if the original design has been degraded by the maintenance. The core of the paper is devoted to the application of the slicing technique to extract all the statements necessary to implement a subset of the interface of a component. This is called interface slicing. For example, let us have a component with interface functions \( f_1, \ldots, f_n \). If we only wish to reuse the functions \( f_3 \) and \( f_4 \), then the interface slicing technique would remove all the code in the component that is not involved in the implementation of \( f_3 \) and \( f_4 \). The interface slicing engine would take as input the whole component and the required interface function. It would produce as output a new working component with only the code to run these functions. To compute such a slice, the authors propose first to build the interface dependence graph i.e. the graph that represents the dependence between all elements in the component. Then the graph is traversed to retrieve all the statement and declarations transitively linked to the interface function to keep in the new module.

Ning et al. presents the use of program slicing to recover reusable components from a legacy Cobol code. This is part of a larger Cobol reengineering environment called COBOL/SRE [Nin93]. First the authors remark that two characteristics of a legacy code may make it difficult to reuse. First a large amount of the code is platform-specific. Second, old programs are typically large and unstructured and the functional units are often interleaved. Generally, these large modules must be broken down to smaller, more cohesive units. This is where the slicing techniques, named program segmentation in the article, come into play. The COBOL/SRE environment make use of the condition-based slicing, forward slicing and backward slicing techniques. The first is somewhat similar to quasi-static slicing: starting from a conditional statement with some values assigned to the variables in the condition, all the statement that are reachable along the control flow path are added to the slice. Forward slicing: this technique retrieves all the statement that can be potentially affected (both from a data-flow and control flow point of view) by the value of a given variable. Finally backward slicing corresponds to the traditional slicing technique. It retrieves all the statement that may potentially affect the value of a given variable at a given location. Then the result of all these slicing techniques may be combined through the union, intersection and difference operators.

The work of Lanubile and Visaggio [Lan97] is aimed at extracting reusable component from legacy code. Starting from the finding that traditional static slicing produces much more statements that strictly necessary, the authors propose a new technique called transform slicing. The main idea is to extract the statements that computes the values of a set of output variables at a given program location \( s \), given the value of a set of input variables. Then, the author define the transform slicing criterion as a triple \(<s,V_{in},V_{out}>\) were \( V_{out} \) is the usual set of output variable whose values are controlled by the statements in the slice. However, in this case, the values of a set of input variable \( V_{in} \) are taken into account. Therefore the definition of the slice is the same as the original one except that for each node, the computation of the relevant set of variables excludes \( V_{in} \). Moreover, when retrieving the conditional statements that influence the execution of the statement in the slice, the authors exclude all the condition statements that influence the execution of the criterion statement \( s \). The reason is that depending on the value of the condition, the latter may not be executed. As we want to extract the statements that implement a function, the condition statement that control the execution of the function should be left outside, in the calling code.

---

16 In the sense of the construction of some higher level abstraction from the implementation.
Definitions: let G = (N,E) be a control flow graph where N is the set of nodes that represents the program statements E is the set of edges that represent the control flow between the statements. SUCC(n) represents the set of node that immediately follow n in the graph i.e. m ∈ SUCC(n) ⇒ (n,m) ∈ E . INFL(n) represents the set of nodes that follow n in G and that may be reached or not depending on n. If INFL(n) is not empty then n is a condition statement or a condition loop statement. Finally, D(n) represents the set of variables defined at node n and U(n) represents the variables used at node n. With these definitions, the intraprocedural\textsuperscript{17} transform slice TrS\textsubscript{c} for the slicing criterion c = <s,V\textsubscript{in},V\textsubscript{out}>, can be defined by the following expression:

\[ \text{TrS}_c^0 = \{ n \in N \mid D(n) \cap \text{TrR}_c^0(\text{SUCC}(n)) \neq \emptyset \} \]

Where TrR\textsubscript{c}^0(n) represents the set of relevant variables at node n. The expression is similar to the original one taken from the work of Weiser. Now, the relevant variables TrR\textsubscript{c}^0(n) = { v ∈ V\textsubscript{out} | n = s } ∪ { U(n) - V\textsubscript{in} | D(n) ∩ TrR\textsubscript{c}^0(\text{SUCC}(n)) = \emptyset } ∪ { TrR\textsubscript{c}^0(\text{SUCC}(n)) - D(n) }

Finally we must compute the set of conditional statements TrB\textsubscript{c}^0 that control the execution of the statements in the previously computed slice TrS\textsubscript{c}^0:

\[ \text{TrB}_c^0 = \{ b \in N \mid \text{INFL}(b) \cap \text{TrS}_c^0 \neq \emptyset \land s \notin \text{INFL}(b) \} \]

The complete set of statements in the slice must be computed recursively, by including, at each step, the nodes that control the statements gathered by the previous step. Then, we can reformulate the previous equations by including the index of the computation step. The corresponding algorithm is run until no more nodes are added to the slice:

\begin{enumerate}
  \item TrR\textsubscript{c}^{i+1}(n) = TrR\textsubscript{c}^i(n) \bigcup \limits_{b \in \text{TrB}_c^i} TrR\textsubscript{c}^0_{s,V\textsubscript{in}(b),V\textsubscript{out}(b)}(n)
  \item TrS\textsubscript{c}^{i+1} = \{ n \in N \mid D(n) \cap TrR\textsubscript{c}^{i+1}(\text{SUCC}(n)) \neq \emptyset \} ∪ TrB\textsubscript{c}^i
  \item TrB\textsubscript{c}^{i+1} = \{ b \in N \mid \text{INFL}(b) \cap TrS\textsubscript{c}^{i+1} \neq \emptyset \land s \notin \text{INFL}(b) \} \}
\end{enumerate}

Then, the authors extend their definition of the transform slicing to a program with many procedures that call each other. The interprocedural control flow graph is built by linking the control flow graphs of all the procedures in the program with edges that represent the call/return control transfers between the nodes of the procedure’s graphs. However, this is not enough. We must also take the mapping of variables between caller and called procedures into account, plus the global variables that cross the boundaries between the caller and the called. Then the authors define a descending slicing criterion to slice a called procedure after having mapped the input parameters of the call and the ascending slicing criterion to slice the caller procedure with the result of the call after having mapped the output parameters of the call.

Discussion: since the technique is used to extract reusable function form some legacy program, the problem is to identify the final statement of the function together with the input and output variable i.e. the slicing criterion. Once this is known then the slicing technique could retrieve all the statements that implement the function. But the finding of the criterion is the hardest and non algorithmic part of the task. Then, the author cites some earlier work where a previous data model recovery phase may help to identify some useful data to be used as a starting point. Finally, this research leaves open the question to identify what

\textsuperscript{17} The computation of the interprocedural slice is more sophisticated as it must take the procedure calls into account.
function to recover from a given legacy program i.e. what could be usefully reused in a new system.

The research of Wong et al. [Won99] targets the localization of program features in the code using another technique called execution slices. In fact, one of the first steps in program maintenance is to locate the code that implements a given feature. Basically, this operation requires the maintenance engineer to understand the code. Then, the authors highlight the two fundamental strategies: complete understanding prior to performing any code modification and as-needed understanding to identify the relevant code as quickly as possible. Although the less risky, the first strategy is often impractical for industrial-size software systems. Often, we are forced to limit ourselves to a partial understanding of the system. In this case, the static and dynamic slicing techniques can help to find the relevant part of the code. However, the author recall that static slicing will typically return more code statements than dynamic slicing and then is less effective. On the other hand, dynamic slicing may consume excessive time and file space. Then they propose to use the execution slices which are the statements executed when running a test case. This technique could use the traces collected during the tests of the system. The authors define the invoking test for a feature f as a test that, when executed, shows the feature f. Conversely, the excluding test is one that does not show the feature f.

Let T be a set of tests, \( \bigcup_{\text{invoking}} \) be the set of program statements that are executed when running at least one of the invoking tests in T, \( \bigcup_{\text{excluding}} \) be the set of program statements that are executed when running at least one of the excluding tests in T and \( \bigcap_{\text{invoking}} \) be the set of program statements that are executed in each invoking tests in T. From these definitions the authors present three strategies to find the code relevant to the feature f:

1. remove from the smallest execution slice among all the invoking tests all the program statements of the largest execution slice among all the excluding tests.
2. \( \bigcup_{\text{invoking}} - \bigcup_{\text{excluding}} \) i.e. keep the program statements that are executed by any invoking test but not executed by any excluding test
3. \( \bigcap_{\text{invoking}} - \bigcup_{\text{excluding}} \) i.e. keep the program statements that are executed by every invoking test but not executed by any excluding test.

Finally the code common to a set of features \( \{F_1...F_n\} \) is simply given by \( \bigcap_{f_i \in \{F_1...F_n\}} \bigcup_{\text{invoking}} (f_i) \)

The identification of the common code of a set of features is important for maintenance: it is the code that may result in side effect when changing it to maintain one of the features.

In their study of some real world program, the authors found the second heuristic to work best for identifying the code that is uniquely related to feature f. But this is true only if we can find some tests that uniquely exhibit f. If this is not the case (if several features are exhibited by any tests) then it might be better to use the third heuristic. Finally, the authors note that the execution slices associated to the invoking and excluding tests for a feature should be as similar as possible in order to filter out as much common code as possible. To gather the execution slices, the authors used a development environment that traces all the statements executed during a given test.

Discussion: rather than exploring all possible tests to find the element of code relevant to some features, the authors proposed to use a few carefully selected tests. They are well aware that this technique may miss some important part of the code, but they insist that their technique must be used as a starting point for studying program features. Moreover, these execution slices may not able to identify all the attributes and data structures used by the
features. If this identification is required then the authors propose that the execution slice technique be complemented by static slicing.

Interestingly, the same kind of approach using test cases has been proposed much more recently to recover the code associated to the features of a program using the formal concept analysis technique, as we will see in the next chapter.

Finally, Harman et al [Har02] used the slicing technique to gather the statements that pertain to a given domain concept. This research rests on the concept assignment technique developed by one of the authors (N. Gold). However, the latter cannot usually produce an executable program that corresponds to the domain concepts identified in the code. Then, starting from the statements produced by concept assignment, they retrieve all the missing statement with a classical slicing algorithm.

2.3 Formal concept analysis-based reengineering systems

2.3.1 Brief introduction to the theory of Formal Concept Analysis.

Let us have a set of objects $O$, a set of attributes $A$ and a relation $P$ between the objects and attributes: $P \subseteq O \times A$. The tuple $(O, A, P)$ is called a formal context.

Let us define the following two operations:

- $\sigma(O)$ returns, for a set of objects $O$, the largest set of attributes that are common to all the objects in $O$: $\sigma(O) = \{ a \in A \mid \forall o \in O : (o, a) \in P \}$
- $\tau(A)$ returns, for a set of attributes $A$, the largest set of objects that own all the attributes in $A$: $\tau(A) = \{ o \in O \mid \forall a \in A : (o, a) \in P \}$

Given a set of objects $O \subseteq O$ and a set of attributes $A \subseteq A$, the pair $(O, A)$ is called a concept$^{18}$ iff $A = \sigma(O)$ and $O = \tau(A)$.

In other words, for a concept $C = (O, A)$, there does not exists any object $o \not\in O$ that would own all the attributes of $A$ and there does not exists any attribute $a \not\in A$ that would be owned by all object in $O$. The set of object $O$ of a concept represents its extent while the set of attributes represents its intent. From these definitions, it is easy to define a partial order on the concepts:

$$(O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2 \text{ or } (O_1, A_1) \leq (O_2, A_2) \iff A_2 \subseteq A_1$$

If $C_1 \leq C_2$, $C_1$ is called a subconcept of $C_2$ and $C_2$ a superconcept of $C_1$. (The bigger the set of objects included the bigger the concept and the bigger the set of attributes the smaller the set of objects in the concept). The set $\mathcal{C}$ of all the concepts of a given formal context together with the partial order $\leq$ form a lattice that could be represented as an acyclic graph. The top element in the lattice, i.e. the superconcept of all concepts, holds all the objects and the set of attributes (possibly empty) that are common to all the objects. This object is called the supremum (or join) and noted $\text{sup } \mathcal{C}$ or $\lor \mathcal{C}$. On the other hand the bottom element, i.e. the

$^{18}$ The notion of concept used here should not be confused with the kind of mental construct we could find in the cognitive science work. It is a formal definition based on binary relations.
subconcept of all concepts, holds all the attributes and the set of objects (possibly empty) that share all these attributes. This object is called the infimum (or meet) and noted $\inf C$ or $\land C$. The supremum of a set of concepts $\bigvee\{C\}$ can be computed by intersecting their intents (set of attributes) and finding the common object sharing all the resulting attributes:

$$\bigvee\{O_i, A_i\} = (\tau(\bigcap A_i), \bigcap A_i)$$

A concept formed from a singleton set of objects is sometimes called an atomic concept. Let $\mathcal{B}$ be a formal context and $o$ be an object in this context. Then, the atomic concept formed from the singleton set $\{o\}$ is computed as: $(\tau(\sigma(\{o\}))$, $\sigma(\{o\}))$. In other words, the extent of this atomic concept is the set of objects that share the same attributes as $o$. Therefore it is not a concept whose extent contains a single object, but a concept whose intent corresponds to all the attributes of a single object.

Once all the concepts are computed for a given context, we may realize that some objects belong to several concepts. Then, a concept partition is a set of concepts whose extents represent a partition of the set of objects of the formal context. The set of concepts $CP = \{C_1...C_n\}$ represent a concept partition with respect to the set of objects $O$ iff:

$$\bigcup_{i} \text{extent}(C_i) = O \land \forall i, j i \neq j \text{extent}(C_i) \cap \text{extent}(C_j) = \emptyset$$

Then, an atomic concept partition is a partition whose concepts are atomic. If the atomic concepts of a given context $(O, A, P)$ form an atomic partition, then the following property holds:

$$\forall o_1, o_2 \in O, \sigma(\{o_1\}) \subseteq \sigma(\{o_2\}) \Rightarrow \sigma(\{o_1\}) = \sigma(\{o_2\})$$

A context for which an atomic partition can be computed is sometimes called a well formed context.

Finally, it is important to note that this theory can be used whatever the objects and the attributes, as long as we could define a relation between them. In the following, we will see the use of this theory for different set of objects and attributes.

### 2.3.2 Module Identification

In December 1999, Siff and Reps [Sif99] proposed the use of the concept analysis technique to identify modules in procedural code. In this work, the central hypothesis is that a good modularization of a given procedural program could be built by choosing the modules in a concept partition of the context whose objects are the functions of the program and the attributes the properties of those functions. Since the extent of the concepts represents non-overlapping sets of functions, the concepts could represent candidate modules for the program.

In this case, the modularization problem is then reduced to the computation of a concept partition of the program, from which a software engineer would choose the “best one”. Then, the core of the paper proposes an algorithm to compute the concept partitions. It also shows how to overcome the problem of the programs that do not have any concept partition but the trivial one\(^{19}\).

\(^{19}\) A trivial partition consists of one concept whose extent is the entire collection of objects of the context.
Although it seems natural to define the functions as context’s objects for procedural program modularization analysis (after all, what we really wants is to group the function into non overlapping sets representing modules\(^{20}\)), the choice for the object’s attribute is larger. In fact, we may use whatever properties seem appropriate to identify concepts. Then, the authors list the following possibilities:

- Global variable usage;
- Dataflow (“use of value flowing from a given statement”) and slicing information (“is part of a given slice”);
- Type inference (“has argument whose type \(t\) is subtype of \(T\)”);
- Disjunctions of attributes.

In some of their examples, the authors used attributes that reflect the way \textit{struct} data types are used (“returns \(s_1\)”, “has arg \(s_2\)”, “uses \(s_3\) fields” where \(s_1\), \(s_2\) and \(s_3\) are \textit{struct} data structures).

The proposed algorithm is based on the prior computation of the atomic concept partition for the context\(^{21}\). However, the authors remark that not all contexts have an atomic partition (they may even not have any partition but the trivial one). In fact, a context \((\Omega, \Lambda, \Pi)\) for which no atomic concept partition exists is one where the following property does not hold:

\[
\forall o_1, o_2 \in \Omega, \quad \sigma(\{o_1\}) \subseteq \sigma(\{o_2\}) \implies \sigma(\{o_1\}) = \sigma(\{o_2\})
\]

In other words there exists some pairs of objects \(o_1, o_2 \in \Omega\) for which \(\sigma(\{o_1\}) \subset \sigma(\{o_2\})\)\(^{22}\). These pairs are called “offending pairs”. To let the atomic partition be computable, the technique is to extend the context\(^{23}\) by adding an attribute as well as the corresponding entries in the context relation such that, in the extended context, \(\sigma(\{o_1\}) \subset \sigma(\{o_2\}) \land \sigma(\{o_1\}) \neq \sigma(\{o_2\})\). The authors then propose two alternative techniques to extend the original context:

1. For each offending pair \(o_1, o_2 \in \Omega\) with \(\sigma(\{o_1\}) \subset \sigma(\{o_2\})\), add a unique attribute \(a_{o_1}\) and a unique entry in the context’s relation so that \(o_1\) is the only object that is associated to \(a_{o_1}\). In other words, \(\tau(\{a_{o_1}\}) = \{o_1\}\).
2. Augment the context with an attribute that represents negative information. For each offending pair \(o_1, o_2 \in \Omega\) with \(\sigma(\{o_1\}) \subset \sigma(\{o_2\})\), choose \(a \in \sigma(\{o_2\}) \setminus \sigma(\{o_1\})\). Then create the attribute \(-a\), called the complement of the attribute \(a\). Finally add the in the context’s relation an entry \((o, -a)\) for each object \(o\) such that \((o, a) \notin \Pi\).

Finally, the authors present the algorithm to compute the concept partition of a context from the atomic concept partition. In the worst case, the number of partition computed is exponential in the number of concepts. Experiments have shown that it may be more tractable to generate partitions from contexts extended by adding a unique identification attribute (alternative (1) above) that it is from contexts extended by adding negative information (alternative (2) above). Although a real world test of the proposed technique is not presented in the paper, the authors are confident that this technique could bring useful results, especially in the chosen C, C++ environment. Moreover they highlight the fact that concept analysis technique somewhat generalize many of the previous modularization techniques. In fact, concept analysis may produce the same results by appropriately

\(^{20}\) For example the authors target the modularization of C code to convert it to C++.

\(^{21}\) The authors highlight that this is the only strategy they investigated although there might exist other ones.

\(^{22}\) Where \(\subset\) represents the proper subset i.e. \(x \subset y \implies x \neq y\)

\(^{23}\) Formally, a given context \((\Omega, \Lambda')\) is an extension of a context \((\Omega, \Lambda, \Pi)\) if \(\Lambda \subseteq \Lambda'\) and \(\Pi \subseteq \Pi'\).
choosing the suitable properties to include in the context. As a counter example, the authors cite results reported by others where the attempt to modularize Fortran and Cobol program have been discouraging. The authors note that, because the original modularization reflects a design decision that is "inherently subjective", it is unlikely that the modularization process can ever be fully automated.

### 2.3.3 Module restructuring

Another interesting use of the theory of formal concept in reengineering is given by the work of Paolo Tonella [Ton01] where it is used to analyze the quality of a module structure of a given piece of software. In this work, a formal context is defined from the set of all program functions, all the data structure of the software and the "access-to" relation. Then, given a concept \( C = (F,D) \), its extents \( F \) represent the set of functions that together access all the data structure represented by its intent \( D \). With such a formal context, it is clear that a concept bears some close resemblance to the notion of module in software engineering if we build a set of concepts such that the intent of any two concepts in this set do not intersect. In this case, the concepts represent a well encapsulated set of modules where the data structure (intent) of any concept cannot be access by the functions (extent) of any other concept.

Then, the theory of formal concepts analysis allows the author to formalize the study of the module structure of a piece of software. For example, it is possible to quantify the encapsulation violation by measuring the number of attributes in the intersection of the intent of any two concepts. Of course, a good encapsulation is not enough to get a well structured software, the level of decomposition is also important. For example, a software with only one concept or module would be perfectly encapsulated, but we would not say that it is optimally structured. Then, a good structure is a tradeoff between the encapsulation level and the decomposition level. Finally, the author explores the cost of restructuring a software system based on the distance between two module structures.

The module structure of a software system can be retrieved based on the notion of concept partition. If a given set of concepts forms a partition of the object (function) set, then every function belongs to one and only one concept (module). However, it is worth noting that a concept partition does not imply a good data encapsulation. Depending on the allocation of the functions among the concepts, we may well have some functions in one module which access the data structures in another module. Moreover, in practical situation, it is not desirable to force all the program functions to form a partition because some function may not access any data structure. These functions would then only be part of a concept (module) whose intent is the empty set. By definition, such a concept would include all the program functions. Therefore the author suggests that the constraint of object partition be released in favor of an concept subpartition. A set of concepts \( CSP = \{C_1,\ldots,C_n\} \) represent a concept subpartition\(^{24}\) iff:

\[ \forall \ i,j \ \ i \neq j : \text{extent}(C_i) \cap \text{extent}(C_j) = \emptyset \]

From the notion of concept partition, the author defines the object subpartition as the set of the extents of the concepts of a concept subpartition. Given a concept subpartition \( CSP = \{(X_1,Y_1),\ldots,(X_n,Y_n)\} \), the object subpartition is \( \{X_1,\ldots,X_n\} \). Then, the latter can represent a possible modularisation of a program.

---

\(^{24}\) i.e. one releases the other condition: \( \bigcup_i \text{extent}(C_i) = O \) . See § 2.3.1
Starting from the fact that a given attribute may be referenced by the functions in different concepts (modules), the author proposes a measure of the encapsulation violation. To define such a measure, we must first assign each attribute to a given concept. The assignment criteria will be the number of accesses from the functions of the concepts. An attribute will then be assigned to the concept (module) having the highest number of functions accessing the attribute (data structure). Then, the count of the encapsulation violations will be computed as the total number of accesses from functions in a module to attributes in another module.

Although the number of encapsulation violation gives a good indication of the quality of the structure of the software, it could not be the only criteria. Again, the number of modules the software is decomposed into is also an important factor. But the author does not provide any specific measure of the desirable decomposition level beyond the mere number of modules in the program.

Finally to compute the cost of restructuring an initial module structure to a target one, the author proposes to measure the distance between the two concepts subpartitions. First, the author remarks that a concept partition \( P \) could be transformed into a new concept partition \( P' \) by the repetitive application of the following elementary transformations:

1. move of one object (function) from one concept (module) to another one.
2. remove of one object (function) from a concept (module) to create a new concept with the removed function as its single element (singleton concept)

Then, the distance between two partitions is the minimal number of elementary transformations that would transform the source partition to the target one. The author notes that this distance only gives a coarse indication of the cost or effort needed to restructure a program. In fact it must be weighted by the cost of an actual move. Finally the author remarks that the restructuring of the modules is not the only action we could take to remove an encapsulation violation. We could also change the code of a function.

### 2.3.4 Linking features to source code

From the study of RIGI and SAR, we see that the main difficulty in the reverse-engineering of a legacy system in the absence of a reliable knowledge source is the identification of higher concepts from the source code. Beside an exhaustive redocumentation that could be very complex, one technique is to start from a given set of functional requirements and identify the locations in the source code where they are implemented. This is the technique explored in the work of Eisenbarth and Koschke [Eis03]. The idea is to identify the implementation of features in the source code, where a feature is defined as "an observable behavior of the system that can be triggered by the user". The authors acknowledge the fact that an exhaustive identification of all the features in the source code may well not be cost-effective. Then, their technique let the engineer focus on a subset of the features only.

Starting from a given feature to locate in the code, the first step is to ask a domain expert to design a system usage scenario\(^{25}\) that invokes the feature (leads to the observable result). The program execution will then be monitored to trace the set of involved computational units (function, class, modules or components, depending on the granularity level required). The authors are well aware that the generation of scenarios with their set of input data to retrieve

\(^{25}\) The notion of scenario used here is not to be confused with the notion of use-case [Jac92]. In fact, a use-case should always be designed so that it provides a result of value to the actor [Jac94]. In this work, the purpose of the scenario is only to invoke some targeted feature that may well not provide any particular result of value to the user. In fact Eisenbarth and Koschke highlight that "[Scenarios] should invoke all relevant features but as few other features as possible to ease the mapping from scenarios to feature and from feature to source code".
all the code associated to a given feature is a complex issue. This is why they insist on the use of the dynamic analysis only as a guide to explore the static structure of the code (the static dependency graph). The final result of the process: the building of the feature-unit map will rest on this guided static analysis of the code.

To build the feature-unit map we must first build a concept lattice where the objects are the computational units of the software and the attributes are the scenarios. A scenario is related to a computational unit if this unit is executed when the scenario is performed. A given concept $C = (U,S)$ is the largest set of units $U$ that are all triggered by a given set of scenarios $S$ and no other scenario could trigger all the units in the set. The top element in the lattice represents the set of scenarios (possibly empty) that invoke all computational units. The bottom element represent the units that are invoked in all scenarios. Then, the lower the concept in the lattice, the less the number of units in the concept (and the larger the number of scenarios).

Now we must compute the mapping from the features to the computational units. Formally, a scenario can be represented as a set of features: $s = \{f_1, \ldots, f_i\}$, where the features of the set are the ones that are invoked when $s$ is executed. As the scenarios are related to computational units, so are the features. To isolate the computational units associated to a given feature, we start by finding a set of scenarios $S$ whose features set intersection leads to the corresponding feature. Then the computational units are given by $\tau(S)$. For example, let us have the following scenarios: $s_1 = \{f_1, f_2, f_3\}$, $s_2 = \{f_2, f_4, f_5\}$. Then, $s_1 \cap s_2 = \{f_2\}$. If, on the other hand, we have $\tau(s_1) = \{u_1, u_2, u_3\}$ and $\tau(s_2) = \{u_1, u_3, u_5\}$ then $\tau(s_1 \cap s_2) = \{u_1, u_3\}$. In a first approximation, we can then say that $\{u_1, u_3\}$ are specific to $f_2$. Starting from this idea, the authors define five categories for computational units with respect to a feature $f$. Let $\Sigma$ be the set of all scenarios. If we define $S_i = \{ s \in \Sigma \mid f \in s \}$ we have, for a given unit $u_i$:

- **Spec:** $u_i$ is **specific** to $f$ iff $u_i \in \tau(S_i)$ & $\forall s \in (S \setminus S_i) \not\exists \not\exists s \in S \setminus S_i \mid u_i \in \tau(s)$
- **Rlvt:** $u_i$ is **relevant** to $f$ iff $u_i \in \tau(S_i)$
- **Cspc:** $u_i$ is **conditionally specific** to $f$ iff $\exists s \in S_i \mid u_i \in \tau(s)$ and $\forall s \in (S \setminus S_i), u_i \not\in s$
- **Shrd:** $u_i$ is **shared** with respect to $f$ iff $\exists s \in S_i \mid u_i \in \tau(s)$ and $\exists s \in (S \setminus S_i), u_i \in s$
- **Irlvt:** $u_i$ is **irrelevant** to $f$ iff $\forall s \in S_i \mid u_i \not\in \tau(s)$

Finally, a feature-specific concept $c_f = (U,S)$ is such that $\bigcap_{s_i \in S} s_i = \{f\}$

If, for a given feature, there is no feature-specific concept in the lattice then we need to design additional scenarios. Fortunately, when all computational units are known in advance (through a static analysis for example), it can be shown that the addition of new scenarios will not modify existing nodes in the concept lattice but only add new ones.

The lattice of concept is built to provide a starting point to the exploration of the static structure of the code and to retrieve the source code that is relevant to a given feature $f$. Once the concept lattice is built, we locate all the feature-specific concept $c_f$ in the graph and identifies all the computational units $u_i$ for which $u_i \in \text{extent}(c_f)$. These represent the starting set of computational units from which the static dependency graph is manually

---

26 The relation between the features and the computational units that implement the features.

27 In fact, there might be more than one $c_f$ for a given feature $f$ in the concept lattice when different computational units are triggered through the execution of a set of scenarios. Recall that the concept are built from the scenarios and the computational units, not the features.
inspected. In fact, due to the likely incomplete input coverage of the scenarios, some feature-related computational units may never be triggered by the expert-designed scenarios. Then, a manual inspection of the code is always desirable, for which additional techniques such as program slicing [Bin96] can be applied.

In summary the feature-specific computational unit retrieval process is made from the following steps:

1. An application expert must design the scenarios that invoke a given feature.
2. The source code is compiled or instrumented to record execution traces. The way to gather execution them depends on the available tools. For example, this could range from the insertion of new statements into the source code to the use of sophisticated debugger options.
3. The scenarios are executed and the corresponding traces recorded.
4. A concept lattice is automatically built from the generated profiles.
5. The concept lattice is interpreted to identify the units that are required for a given feature.
6. Starting from this set of units, the static structure of the program is investigated to retrieve all the units that are related to the feature.
7. If necessary, the set of scenarios is expanded to increase feature differentiation, and the process is started again at step 3.
8. Once all the feature-specific units are retrieved, they are recorded in the feature-unit map.

2.3.5 Discussion

For the static dependency graph analysis, the authors use an extension of the Rigi system [Mul93, Won98]. But, as mentioned before, the direct analysis of a dependency graph can be very hard. Then, the dynamic technique proposed here represents a very efficient way to “prune” the search space. However, according to the authors, some problems with this technique could come from the duplicated code. Another problem comes from the parameterized scenarios i.e. scenarios that execute different computational units based on the value of input parameters. It might then be hard to design the right set of scenarios to invoke a given feature. Although the size of the concept lattice is generally much smaller than the corresponding static dependency graph, it might still be a challenge to display. In fact, to overcome the screen size limitation, the authors used a large printer in one of their experiment with this technique. Finally, it is worth noting that this technique bears some close resemblance with the execution slicing technique of Wong et al [Won99]. The latter is a technique that gathers the statements executed when running a test case. Then, the statement specific to a given feature are identified by running tests cases that exhibit and do not exhibit this feature and by later performing some simple set-operations. If the starting idea is the same in both works, the analysis of the output of the execution trace is much more sophisticated in the work of Eisenbarth and Koschke [Eis03].

2.4 Other trends

One of the early environment for reverse-engineering, the DECODE system [Qui95], is based on three components: an automated plan recognizer, a knowledge base for recording extracted design information and a structured notebook for editing and querying this design. The plan recognizer works bottom-up by trying to recognize standard code patterns in the source code. Once a plan is found it is stored in the knowledge base. Then the programmer uses the notebook to inspect the program source code: whenever the

28 The author cite some earlier work of others where the recognizer would work top-down. However it has been found that the top-down technique does not scale well.
programmer recognizes the purpose of some arbitrary code segment, it maps it to an object or an operation in a programmer-created object-oriented hierarchy. First, the programmer is provided with two windows: the code browser and the design editor. He will then record his understanding of the code by interactively adding new conceptual design element and linking them to a recognized plan or to some highlighted portion of the code.

The plans are stored together with some heuristic information to help their recognition in the code. Once a plan has been recognized, the system tries to connect it to higher level programming concept such as opening a file, validate a record, and so on. Finally, the notebook allows conceptual queries to be issued. Here are some examples of such queries. What is the purpose of this piece of code? Where is the code corresponding to a given design element? What are the lines that are not yet explained? What are the unconnected design elements? The limitation of this system is that it is very difficult for people other than the systems’ designers to formulate and enter plans. Moreover there is no facility for the top-down analysis of programs. Finally, it is somewhat difficult to relate “delocalized” program lines belonging to the same design element.

A second system that builds a mapping from the high level concepts to the corresponding elements of the source code is presented in the work of Nicolas Gold [Gol01]. Starting form a knowledge base of items and activities in a domain, his system, called HB-CA, finds the corresponding code elements using a plausible reasoning system based on a neural network technique. This network analyzes the source code to identify names and comments and automatically binds the relevant segment of the code to the concept in the knowledge base. Although the author reports some impressive performance on small programs the recognition capability seems to fall rapidly beyond 1000 LOC.

Another trend in software reengineering is to analyze the source code of the legacy system to find occurrences of design patterns. For example, Ascencio et al. [Ase02] describe a system, called Osprey, which let the user detect the use of most of the Gamma’s design patterns [Gam85]. In Osprey, each potential pattern is coded together with one or more recognizers. The latter is a procedure written in some set-theoretic language that will check the code for the occurrence of the pattern. Their approach seems to have been rather successful at detecting pattern in source code written in C++.

On the other hand, Tahvildari and Kontogiannis [Tah02a, Tah02b, ] re-examined the 23 GOF design pattern in a reengineering perspective. In fact, the goal of their work is to enhance the quality of some object-oriented code where the quality is defined in terms of non-functional requirements. Then, they classified the GOF design pattern according to their impact on the quality of the code. This can be used by a maintenance engineers to choose a pattern to restructure the code. Here we see some similar ideas as expressed in the horseshoe model [Ber99b] where the improvement of the quality attributes of the architecture is the driver of the re-engineering work. However, in this case, the technique works at the lowest conceptual level of the program and does not address the meaning of the program. Based on this previous work, Tahvildari later proposed a Quality-Driven Reengineering method that focuses the restructuring of the classes whose quality has deteriorated. First, the target quality of software must be expressed, for which the author uses the formalism of soft goals

---

29 The system has been used to analyze COBOL II source code. Apparently, only the procedure division (function and procedures) is analyzed but not the data division (declaration of the data structures).

30 A soft goal represents a non functional attribute of the software. The soft goals are represented a tree. Each of a node’s child represent an attribute that contributes to the achievement of the attribute represented by of the node. The root of the tree represent a classical software quality attributes such as maintainability, performance…
code is restructured by the application of the design patterns chosen according to their impact on quality. Finally the quality of the resulting software is globally reassessed.

The idea of patterns has also been applied to document the knowledge necessary to reconstruct a software architecture. For example, Stoermer et al. [Sto02] documented six patterns to codify the general techniques for architecture recovery. The View-Set pattern expresses the need to identify the relevant views of the system under analysis to fully understand its structure. The Enforced Architecture pattern represents the task to compare the as-designed architecture to the as-built architecture. The Quality Attribute Changes pattern represents the problem to link the quality attributes (in the sense of [Bas03]) to the architectural patterns. The Common and Variable Artifact pattern represents the task to identify the common and variable parts in a line of products. The Binary Component pattern focuses on the problem to reconstruct the architecture from the binary code of some commercial component. Finally the Mixed Language pattern addresses the problem to analyze a system whose components are written in many different languages. From these patterns, the authors have analyzed some of the current reengineering frameworks and assessed their support of these patterns. The finding is that none of them supported all the patterns that nonetheless represent fundamental techniques of reengineering.

The approach taken by Gall, Weidl and Klotsch [Gal98, Gal99, Klo96] is to work top down. The first step in their technique is to build an object model of the main domain abstraction implemented in the code, using all documentation and expert advice available. Then the source code is analyzed to find candidate software objects. Finally the two sets of objects are matched using a similarity factor based on syntactic (name) and semantic (data type) properties. Human expertise is required to solve the many ambiguities that arise during this process. Finally, a binding table is created that records the association between the objects of the domain abstraction model and the recovered objects form the source code. However, the amount of code that the system could match against some domain concepts is limited. In fact many program elements are left without finding a suitable mapping. The immediate conclusion we can draw from this work is that the code associated to the entity objects in a given program only represents a limited part of the total amount of code.

Recently, Marcus proposed semantic technique to drive program analysis [Mar04]. His approach is based on machine learning and document indexing techniques (the latent semantic indexing technique LSI) applied on the source code and the associated documentation (i.e. user manual). Therefore, this method is applicable only if there is enough documentation on the program or its use. This is generally not the case for legacy system.

Starting from the ideas of Markus, Kuhn at al. [Kuh05] built a tool called Hapax to identify clusters in the source code using the same latent semantic indexing technique (LSI) on the source artifacts. The central hypothesis is that the artifacts that use similar terms are semantically related. The inputs of the indexing mechanism are the identifiers and the comments in the source code. Then, a term-document matrix is built where a document represents some subset of the source code (module, class, method). The LSI technique is used to determine the semantic similarities between software artifacts. The result is displayed as a correlation matrix of the documents. The stronger the correlation, the darker the dot at the intersection of the corresponding line and column. If the matrix is sorted according to the result of the clustering algorithm, then we can identify clusters of strongly correlated documents (software artifacts). Finally, the authors plan to apply this technique to the recovery of software architecture.
3. Goal of the project

3.1 Introduction

The goal of this research is to design a formal process for information system reengineering. Software reengineering and program understanding has been a hot topic for at least a decade. It is even more so today due to the increasing concern about the cost of software investments and maintenance. From year to year, maintenance takes an ever increasing part of the total software cost. Some study even shows that, in the 2000’s, it reached roughly 90% of this cost [Kosk].

Regarding these costs, the temptation might be high to redevelop the system from scratch in order to “clean up the mess”. Although this could have been an option some decades ago, in the good old world of small standalone applications, in today’s highly integrated and complex systems it is barely economically feasible. In fact, the software size of most of today’s real world information systems amounts to thousands of KLOC$^{31}$. Moreover, as it is widely known, the documentation of the software systems are often outdated if present at all. Even in the case of the redevelopment of such a system from scratch, the developers would need to investigate the current system to rebuild its specification, as most business rules are deeply buried into the software algorithms. In other words, they would at least reuse the knowledge that has accumulated during the life of the system and which is embedded into the system [Abb93].

If the software system must be restructured rather than redeveloped from scratch, it is clear that any technique that could improve the efficiency of the maintenance process will have a huge economic impact.

So far, many techniques have been proposed to reengineer, reverse-engineer and re-architect legacy systems [Mul00]. In the situation were no reliable documentation on the software structure and function is available, the task of understanding the legacy system may be tremendously difficult. One of the key problems is to map domain concepts to software artifacts (the well-known concept assignment problem [Big94]). This mapping problem is immense in general and, without the help of some reliable source of information, might sometimes be economically unbearable.

Starting from the (rather strong) hypothesis that the initial specification for the software was logically consistent$^{32}$, we can split the maintenance in three broad categories: programming errors or inefficiencies, technological/environmental changes and specification change. The first category includes all software problems due to inadequate implementation of the software specification or to the inadequate evolution of the software$^{33}$. The second category includes the situations where the software must be implemented in another programming language / environment, must use another framework or be run on another platform or hardware. The driver behind this kind of maintenance is often the technology obsolescence. In these first two maintenance categories, the software specification is left unchanged. In the last category, the software is working properly with respect to its current specification and no technology obsolescence forces the software to be maintained.

$^{31}$ For an example of the software size of some real world software system see [Jon99]. On the other hand, if one take the Windows operating system as another example, its size went from about 3 million LOC in 1990 (Win 3.1) to 40 million LOC in 2002 (Win XP) [McG04].

$^{32}$ In fact, the inconsistency or incompleteness of the software specification can be considered as a bug [Bei90].

$^{33}$ this is the case when the maintenance of the program have produced side effects that ruined some quality attribute [Bas03] of the software.
3.2 Business driver

When some specification change request is presented to the IT team, it is very important for the team to understand the business driver behind the request. In the same way as a small error in the output of some accounting program could be the visible symptom of badly designed algorithm, a small specification change request might be the visible symptom of a larger process/IT misalignment. Then, moving up to the level of the business processes could help the developer to forecast future maintenance requests. Although the maintenance due to programming errors and technological/environmental changes has been extensively covered in the literature, much less has been done on the analysis of business process changes’ impact to maintenance. The steps of such an approach are illustrated in the following figure (Figure 10).

Starting from the understanding of the business process changes, the project team must find the ways to realign the software with the current business. It could be a mix between the restructuring of some parts of the current system and the complete rewrite of other parts. Then, the redocumentation of the business processes supported by the software is essential.
### 3.3 Quality driver

#### 3.3.1 Quality attributes

The change to the software might also be motivated by the enhancement of its quality attributes. In short the quality attributes [Bas03] are all the features of a software system which, beyond the functionality of the software, assure that the software will satisfy all its stakeholders. Then they are sometimes called the non-functional qualities of the software. The most frequently cited quality attributes are:

- Maintainability
- Portability
- Performance
- Security
- Usability
- Availability
- Testability
- Flexibility

These attributes are mainly influenced by the architecture of the software [Bas03]. In fact, due to the equivalent power of most popular programming languages, it is clear that the functional result of some piece of software is not influenced by the way the program is written. But features such as performance, flexibility, portability are fundamentally linked to the structure of the program hence its architecture. For example, it is much easier to port an application where the platform-dependent elements are grouped in a well defined layer than if these elements are scattered. Generally speaking, it is not easy to specify the level of a quality attribute for some piece of software since many of these attributes do not have widely accepted measurement techniques. For example, what does “high maintainability” mean? To solve this problem, the Software Engineering Institute designed a framework based on the notion of scenarios. At least, this technique let the engineer specify the measurement conditions and metrics for the quality attribute as well as the expected value [Bas03] (see figure below).

![Quality attribute measurement framework](image)

In this figure (Figure 11), the *artifact*: is the software element whose quality attribute must be measured. Often, this corresponds to the whole system. The *source* is the person or system that interacts with the artifact to test its qualities. The *stimulus* is the action performed by the source, or the information inputted by the source, to the system to test it. The *environment* represents the state of the system and its context during the experiment. For example, we could test the system in normal mode or degraded mode. The *response* is the action taken by, or the behavior of, the artifact after the application of the stimulus. Finally, the *measure* is the expected value of the response, together with the metrics to be used.

---

34 Here we implicitly make the hypothesis that the functionality is correctly programmed and that it conforms to the specification.
Example of scenario to specify the modifiability of the system

Source: Developer
Stimulus: Need to modify some behavior of the program
Artifact: Whole software system
Environment: System in normal operation, in production.
Response: The modification must be effective for the targeted behavior and must not have any side effect on other behavior. The system must be tested and put back in production.
Response measure: Two days maximum delay between the maintenance project start and the system is back in production.

3.3.2 Architecting software for quality

Once the scenario of the quality attribute for the software is specified, the system can be architected to maximize the chances that it will conform to the desired value. These architectural solutions to the quality attributes are called “tactics” [Bac02, Bas03]. They differ from the idea of architectural patterns [Buc96] in that they provide a “style” of architecture that would fit the task but not the schema of the actual components. It is worth noting that the qualities of a system are somewhat dependent on each other. For example, to enhance the portability or maintainability of some program we may accept to lower its performance. The art of architecting a software system to reach some desired level for a set of quality attributes implies to make tradeoffs [Kaz00].

3.4 Mixed driver

In real situations, the drivers behind a reengineering initiative are often mixed: the business process must be adapted, the software system is evaluated as weak with respect to some quality attribute and/or some programming errors must be corrected. Then, rather that doing a small scale maintenance, the management could decide to restructure the software so that it will be more flexible and the company will be more reactive to changes in the future. However because the previous maintenances of the system may have ruined the system’s architecture, it might be the case that the current system cannot be restructured easily. Then first step is to find the software elements that correspond to the business tasks. Next one should analyze the structure of these elements to evaluate the cost of the restructuring work. Depending on the cost, the management may well decide to abandon the full restructuring project and limit the reengineering task to the extraction of reusable components or, even worse, to extract the knowledge from the program to rewrite it from scratch. This is why the scope of a reengineering project may well change during the course of the project.
4. Program comprehension and reengineering

The field of program comprehension is aimed at designing theories and models to explain how programmers proceed to understand programs [Sto05]. These theories and models, in turn, allow the development of tools to help the software engineers to understand the programs they have not written themselves. But the key question here is what “understand a program” really mean? To answer this question without diving into endless philosophical discussion, we borrow the first paragraph of the famous article by Biggerstaff, Mitbander and Webster on the “Program Understanding and the Concept Assignment Problem” [Big94]:

A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.

The “Concept Assignment Problem” represents the search for the implementation in the software of some human oriented domain concepts. In the following text, I will use the term domain concept to mean the human mental representation (concept) of a domain object or operation. Accordingly, I will call the domain model the set of domain concepts together with their relationship. When trying to understand a program, the concept mapping goes both ways:

- From the domain concept to the software, when trying to locate the realization of a given domain concept in the software.
- From the software to the domain concept, when trying to identify the domain concept a given programming construct can be associated to.

Informally, I will call the domain semantics of a given program structure, its mapping to the domain concepts and relationships.

Although they have their own research communities and dedicated scientific conferences, the fields of software maintenance, software comprehension and software reengineering seem to overlap to a large extent. Historically, the research papers on software reengineering somewhat ignored the domain semantics of the structure they try to recover. For example, many techniques have been published to discover components in source code and to recover the architecture of programs. Unfortunately, these papers often do not comment on the domain semantics of these recovered components. However, it seems obvious that not all recovered components organizations are equivalent from the point of view of reengineering. If we keep in mind the goals of a reverse-engineering initiative, then the structures that let the engineer better “understand” the legacy software will improve the reengineering process. In summary, the techniques of software understanding and software re-engineering must necessarily work in concert.

Definition: a meaningful architectural description is one that lends itself to an easy mapping to the domain concepts. In other words, an architectural description is only as good as the mapping it lends itself to.

This definition does not mean that all architectural entities should be mappable to some domain concept, but that the architectural description will allow such a mapping to be possible at some architectural level (or equivalently at some level of granularity). The next paragraph will bring some formalization of this concept.

---

35 Considered in this context as mere program elements aggregates.
36 This is obviously more so when the goal of the reverse-engineering initiative is to understand the software!
On the other hand, a meaningful architectural description will help to understand a software system by reducing its apparent complexity. The trouble with the idea of software architecture is that in most of the situations it is not represented at the code level. To illustrate this problem, Kazman & Carrière even speak of a "shared hallucination" to qualify the quest for software architecture in legacy systems [Kaz97]. In fact, the architectural level of software description is in the head of the designer and, sometimes, in the technical documentation. When confronted with some unknown software system, how can an engineer know that some clever grouping of the software artifact will produce some meaningful description of the architecture? In the case of the legacy systems, it may even be the case that no architecture was designed in the first place! Therefore would any search for the system’s architecture be relevant at all? 

In fact it is relevant. The lack of architectural investigation of the legacy systems to reengineer has been found to be one of the 10 key reasons for the failure of the reengineering project [Ber99]. The other reasons are largely managerial & organizational. For example, the finding that a software system is too messy to have any chance to be reengineered is a valuable result in itself. In this case, some authors would say that this system is too complex to be reengineered. However the assessment of the complexity of a program is generally done on a syntactical basis. On the other hand, it is well known that a system can be more or less maintainable depending on the expertise of the maintenance engineer. This is a clear sign that syntactic complexity is not sufficient to evaluate the maintainability of a program. A better evaluation of the complexity of software would explicitly take into account the complexity of the domain. This is yet another reason why the modeling of the domain plays an important role in software reengineering.

### 4.1 A simple model of program structure comprehension

In order to guide the search for an architectural description of a software system, we must provide some formal definition of program structure comprehension. But we do not have the ambition to provide a general model of program understanding. We only target the comprehension of a program structure in the context of the reverse-engineering of information systems. In particular, this model will not address the problem of algorithm understanding. In fact, our model will formalize the relationship between the software domain and the problem domain. Of course, many authors have recognized the importance of such a relationship in program understanding [Sto05, Til96, Big04] but the proposed mapping is always informal.

### 4.2 Basic definitions

**4.2.1 Program model, domain model**

In this report we focus on legacy information systems only. Then, this work may not be fully applicable to the other categories of software systems. On the other hand, a preliminary, albeit trivial, hypothesis is that the software system to be analyzed does some useful work for some category of people. If this is not the case then, of course, there is no meaning to investigate at all.

**Hypothesis:** a program is written to solve some useful business or engineering task of which the developer has a mental representation37.

---

37 Here, we restrict ourselves to the analysis of a complete program and not some reusable, task independent, component.
In the tradition of knowledge engineering [Nil80], we consider the mental representation of a business or engineering domain as being built from a set of domain concepts and relationships between these concepts. The nature of these concepts will not be detailed any further\textsuperscript{38}. We only need here to admit that these concepts and their relationships are identifiable in the domain. On the other hand, their relationship are not restricted to static associations like the famous “part-of” (composition) or “is-a” (generalization), but may include any relevant “dynamic” ones such as: “produces”, “causes”, “computes” and the like. Especially, as we will see later in this report, the domain models we build represent business processes, tasks and object responsibilities.

Definitions:

1. A domain model $D$ is a pair $(C,K)$ where $C$ is a set of domain concepts and $K$ a set of relations\textsuperscript{39} over these concepts.
2. A program model $P$ is a pair $(E,R)$ where $E$ is a set of program elements and $R$ a set of relations between these elements.

By program element we mean any non-empty set, possibly a singleton set, of programming artifacts\textsuperscript{40} or program elements\textsuperscript{41}.

Property: let $P = (E,R)$ be a program model. Program elements must not overlap i.e.:

$$\forall a,b \in E, \ a \neq b \Rightarrow a \cap b = \emptyset$$

We can now give a precise, yet intuitive, meaning to the notion of level of granularity of a program model.

Definition: a program model $P_1 = (E_1,R_1)$ is considered at a finer level of granularity than a program model $P_2 = (E_2,R_2)$ iff the following three rules apply:

1. $\bigcup_{e \in E_1} = \bigcup_{k \in E_2}$
2. $\forall e_1 \in E_1, \exists e_2 \in E_2 \mid e_1 \subseteq e_2$.
3. $\exists e_1 \in E_1, \exists e_2 \in E_2 \mid e_1 \subset e_2$.

An element $e_2 \in E_2$ is called an aggregate of elements of $E_1$ iff $\exists e_1 \in E_1 \mid e_1 \subset e_2$.

This lets us define a partial order on the program models that we call the granularity-order relation: if $P_1$ is at a finer level of granularity than $P_2$, we write: $P_1 < P_2$. In this case we also say that $P_2$ is at a larger level of granularity than $P_1$.

Definition: two program model $P_1$, $P_2$ are adjacent in the graph defined by the granularity-order relation iff:

\textsuperscript{38}The paper of Rajlich contains an interesting discussion on the notion of mental concept [Raj02]. In particular, he highlights the difficulty to locate concepts in legacy source code.

\textsuperscript{39}Following the knowledge representation terminology, one could also have called $D$ the universe of discourse, i.e. the set of objects and relationships about which knowledge is being expressed [Gen87].

\textsuperscript{40}By programming artifact, we mean any kind of programming language statements or constructs as well as component of the programming environment such as files, records or directories. On the other hand, depending on the grouping of programming elements, these sets may have names such as class, component, module, file, and directory. But the definition does not restrict itself to such “standardized” sets.

\textsuperscript{41}This definition is voluntarily recursive to allow the elements to be represented at different levels of granularity.
• $P_2 < P_1$.
• $\neg \exists P_3 | P_2 < P_3 \land P_3 < P_1$.

### 4.2.2 Discussion

It is important to note that we can build several models of a given program at the same level of granularity. But these models will not be comparable in the graph defined by the granularity-order relation. Of course, not all arbitrary grouping of program elements will lead to some useful program model as we will see later. Similarly, a given domain may have multiple models at the same level of granularity depending on the person considered [Raj02].

### 4.3 Interpretation

The key concept of our definition of program structure understanding is the notion of interpretation, borrowed from formal logic. In fact, this concept is used to provide the semantics of logical expressions [Gen87].

**Definition:** an interpretation $I$ is a mapping from the elements of a program model $P = (E,R)$ to the elements of the domain model $D = (C,K)$ that has the following properties. If we write $I(j)$ the mapping of an element $j$ then:

- If $e \in E$ then $I(e) \in C$
- If $r \in R$ then $I(r) \in K$

**Definition:** we say that an interpretation $I$ from the elements of a program model $P = (E,R)$ to the elements of the domain model $D = (C,K)$ is valid if and only if:

- $\forall a, b \in E, \ a = b \Rightarrow I(a) = I(b)$
- $r(e_1, ..., e_n) \in R \Rightarrow I(r)(I(e_1), ..., I(e_n)) \in K$

**Definition:** the reverse interpretation $I^{-1}$ of an interpretation $I$ is a mapping from the elements of a domain model $D$ to the elements of the program model $P$ such that, for any element $x$ of $P$:

$$I^{-1}(I(x)) = x.$$

### 4.4 Understanding Program Structures

#### 4.4.1 Understanding programs

From the definition of the interpretation between program models and domain models, it is now easier to provide an intuitive definition of what we mean by program structure understanding. First, it is worth noting that a program structure, or the structural aspects of program architecture, is nothing else than a program model\(^{42}\).

**Definition:** we will say that someone or some artificial agent understands a program structure at a given level of granularity iff he can build a valid interpretation $I$ as well as the reverse interpretation $I^{-1}$ between the program model and the domain model at this level of granularity.

---

\(^{42}\) In their widely accepted definition of software architecture [Bas03], Bass et al. add to these structural aspects the externally visible properties of the program elements such as their performance, shared resource usage. However, in a first approximation, these properties can be ignored when trying to understand the structure of the software system.
In this definition, the existence of an inverse interpretation is important. It avoids an interpretation to map all elements of the program model to a single element in the domain model.

4.4.2 Discussion

It is important to note that, in this context, the notion of understanding is fundamentally linked to the domain model. In other words, whatever the cleverness of the observer, if the domain model is not known then there is no hope to understand a program structure.

From the above definition, it is clear that the task of understanding a program structure implies the building of both a model of the domain and a model of the program such that an interpretation (as well as the reverse interpretation) can be built. On the other hand, this understanding will be restricted to the level of granularity for which the interpretation is built. Moreover, the existence of an interpretation at some given level of granularity does not mean that another interpretation can be built at some other level of granularity. Of course, the more complex the mapping between domain and program models the more difficult the understanding of the program structure. In general, an interpretation could be arbitrarily complex, depending on the chosen program and domain models. Intuitively, the more different the structure of both models, the more difficult to find an interpretation. Another consequence of this definition is that any change to a program that impacts its program model at any given level of granularity runs the risk of breaking the interpretation of the engineer then disturbing his understanding.

4.4.3 Meaningful architecture

Informally, an architectural description is a program model at some level of granularity. The above definition of program understanding let us propose a more formal definition of the meaningfulness of such a description.

Definition: an architectural description is meaningful if a valid interpretation I as well as the reverse interpretation $I^{-1}$ of this architectural description can be built.

In other words, a meaningful architectural description is one that effectively helps a maintenance engineer to understand the program.

4.4.4 Understanding complex structures

According to Simon [Sim69], to understand a complex system we must be able to decompose it hierarchically such that:

- The short run behavior of each of the component subsystems is approximately independent of the short run behavior of the other components.
- In the long run, the behavior of any one of the components depends only in an aggregate way on the behavior of the other components.

In fact, when reverse-engineering a complex system, we must build a hierarchy of models and interpretations such that the quantity of information to handle at each granularity level is

---

43 In particular the lowest levels of granularity for software, the statement level, will not lend itself to the easy building of an interpretation. The understanding of the program at the statement level must clearly rest on other techniques.

44 Here the term complexity must be understood in the information theory sense as the quantity of information of the mapping.

45 An industrial size software system can easily be considered a complex system [Til96].

46 It is worth noting that these two rules are close to the classical principles of encapsulation and information hiding in software engineering.
limited. In other words, to understand a complex system, it is not enough to build a single interpretation in isolation. Rather, we must build a set of interpretations that target the different granularity levels of the models. For this hierarchical decomposition to work in practice, a program model at a given granularity level must be compatible with the adjacent program models. This is represented by the following two properties.

Property: (compatibility of aggregates) let us have two adjacent program models $P_1= (E_1,R_1)$ and $P_2= (E_2,R_2)$ with $P_1 < P_2$, then we have:

\[
\forall r_1(e_{11},...,e_{1n}) \in R_1 \exists r_2(e_{21},...,e_{2k}) \in R_2 \mid
(\forall e_1 \in \{e_{11},...,e_{1n}\} \exists e_2 \in \{e_{21},...,e_{2k}\} \land e_1 \subseteq e_2) \land \bigcup_{e \in \{e_{11},...,e_{1n}\}} e = \bigcup_{e \in \{e_{21},...,e_{2k}\}} e
\]

This property means that for every relation at a given granularity level, there must be a relation at the adjacent larger granularity level such that each element of the relation at the finer granularity level is identical to, or included in, an element of the relation at the larger granularity level.

Property: (representativeness of aggregates) let us have $P_1= (E_1,R_1)$ and $P_2= (E_2,R_2)$ two adjacent program models with $P_1 < P_2$ and let $I_1$ and $I_2$ be their respective interpretations, then we have:

\[
\forall e_1 \in E_1, e_2 \in E_2 \mid e_1 \subset e_2 \Rightarrow I_1(e_1) \subset I_2(e_2)
\]

This is an important property: it means that any aggregate in the program model should be interpretable as a higher level concept in the domain model\(^47\). This property will drive the search of components or clusters of program elements during reengineering.

From these definitions and properties, it is now clear that the notion of concept assignment [Big94] is not limited to code statements but must also be extended to every level of granularity.

4.4.5 Discussion

During program design the domain model is generally known by the customer who tries, mainly in the Inception phase\(^48\), to teach it to the developers. Then the developer builds a program whose model’s interpretation should match his understanding of the domain model. Of course, nothing guarantees that the domain models of both the customer and the developer will be the same. If the difference is too big, then the program may show some strange behavior to the customer.

On the other hand, because of the deterioration of the software architecture due to the maintainences, the boundaries of the legacy system “components” may not be very clear cut. In this case, any technique to identify the structure of the system based on the principle of good components encapsulation and decoupling (hence bottom-up) may well provide discouraging results (see [Sif99] for an example of this).

\(^{47}\) In the absence of a formal definition of domain concepts, the inclusion of concepts will be understood as the inclusion of their extensions.

\(^{48}\) In the Unified Process parlance [Jac99].
4.5 Building a domain model and its interpretation

According to our model of program structure understanding, the reverse-engineering process of a legacy system should build:

- A set of program models;
- A set of domain model;
- An interpretation between both sets of models.

In fact, the reverse-architecting of a software system and the understanding of this system are the two faces of the same coin. However, when dealing with a legacy system, we often lack both models. When the unique source of information for the reengineering of a system is its mere source code, the search space of program and domain models is so huge that a simple trial and error process is bound to fail but for trivial programs\(^\text{49}\). This problem can be seen as an instance of chicken and egg syndrome: to manage the complexity of the system and have a chance to understand it we must know its structure. But to recover the structure of the system we should understand its working. To escape from this paradox, we propose to work first at the business process level: this is the level of understanding that can be reconstructed from the users of the system. Starting from this idea, we adapted the steps of the Unified Process [Jac99] to our reengineering method based on program understanding.

\(^{49}\) i.e. a program for which the domain model can be built by common sense.
5. Business modeling and reverse engineering

5.1 Introduction

Following the link we have drawn between reverse architecting and program understanding, the fundamental problem is to partition the set of software artifacts into subsets that are meaningful and to show the relationship between these subsets. The first difficulty is to identify the partition criteria. Moreover, because the program may have undergone much maintenance its original architecture, if present at all, may have changed.

The granularity level of the software structure to recover should be set according to the level of understanding we want to achieve. Clearly, our goal is to recover the structure of the software at the level of the components and the role they play in the overall working of the program. In other words, the target of our reverse architecting effort is to be able to answer the following questions:

- What is the purpose of a given component i.e. to what domain level concept this component could be associated to?
- Where is a given business level function, action or concept represented and manipulated in the code?

On the other hand, the impact of maintenance on the structure of a program depends on the level of granularity considered. For example, a maintenance that changes the way an algorithm computes a result has a clear impact on the program structure at the level of the program statements. Then, at this level, the maintenance is “structure damaging”. On the other hand, if the algorithm is encapsulated in a component whose interface is not changed by the maintenance then, at the granularity level of components, the maintenance is “structure preserving”.

5.2 Size of the recovered components

In their definition of the software architecture Bass et al. [Bas03] speak of “software elements” to name the components of the software architecture. However, since the elements have “externally visible properties” we can infer that such an element is a container (or, to use a term that is closer to the current reverse-engineering literature, a “cluster” [Wig97]).

In this work, we do not look for the exact way a given output is computed but only where, in what component or module, it is computed. Of course, one key problem is now to define the minimal size of the components to consider. This problem can be solved easier if the reengineering method works top-down than if it works bottom-up: the large grain components are analyzed before splitting them into pieces. On the other hand, the size of the components will fundamentally depend on the scope of the reverse engineering project. If the goal is to restructure a system to improve some quality attribute, then the description of the software structure should go to the level where this quality attribute can be handled. On the other hand, if we want to reuse some legacy component in another environment then we may have to dive deeper into the structure.

The top-down approach also has the advantage to limit the complexity of the reverse-engineering task. To make an analogy with the ideas of Simon in complex system understanding [Sim69], we will limit the search to the “long run behavior” represented by the

50 Here the notion of component is to be taken in a broad sense: any meaningful aggregates of program elements 4.4.
components and their interactions. We will not work at the algorithmic level that corresponds to the “short run behavior” of the components.

5.3 Basic hypothesis

Often, the original developers of a legacy information system to reengineer are not available to provide information on the original structure of the program. On the other hand, even if some of them are still available, their vision of the system may be narrowly focused on their area of expertise, missing the big picture. A good perspective on the legacy system is often to be found on the customer or end user side. Although they may have a narrow view of the programs [Abb93] they are usually well aware of the business context and business relevance of the software. They normally know what kind of information must be inputted in the system and when. Although they generally cannot explain the inner working of the software, they usually know the purpose of the program they use and the business justification of the computation. Then, by gathering the information of the system usage by all the involved people, we can reconstruct the sequence of processing at the business level. In short, we can build a representation of the business process the system is intended to support.

On the other hand, when a software system is maintained to adapt it to some specification change (§3.1) then it is likely that the maintenance will be centered on the business tasks performed with the software. Consequently, after several maintenances, the software structure recovered by following the boundaries of the business tasks will likely be more stable than any other structure that would cross the implementation of several business tasks. This is yet another reason why our methodology will focus on the business tasks.

5.4 Architecture views to be recovered

Software architecture is represented through different views of Kruchten [Kru95]. According to our model of software comprehension, we seek an architectural representation of some legacy software that will help us understand where, in the source code, a given set of high level business function is implemented. Moreover we must cluster these functions by business tasks so that the realignment of the system with the business processes is possible (§3.2). Usually, the reverse-engineering tools and methodologies mainly target the static structure of the code. For example Krikhaar [Kri99] developed a formal method to represent the static structure a program, which corresponds basically to the “logical” and “development” views of Kruchten.

Although the target of a reengineering project is mainly the logical view, the development view could bring some important information on the initial structure of the code. If the source code of the software is organized as directories, libraries, packages, program files and/or classes, the corresponding hierarchical structure must have a meaning. At least it was designed to control the complexity of the software by its developer. These elements form what we call the visible high level structure of the system.

In some situations, the static structure of the code must be complemented with the analysis of the working of the programs at run time. This is the case, for example, when some components are dynamically added at run time [Yan04]. In this case, the process view of Kruchten may bring some valuable clue to the maintenance engineer.

51 For a discussion on the role of the views in documenting software architecture see [Bas03]
5.5 Business process terminology

Before describing the use of business modeling for reengineering, it is useful to clarify some terminology. A business process is “a collection of inter-related work tasks, initiated in response to an event that achieves a specific result for the customer of the process” [Sha01]. The two important elements of this definition are: “Specific result” and “customer”. The former stresses the fact that a process should focus on a goal which represents the purpose of the process. Such a result could be a product or a service and must be clearly identifiable and countable. This prevents the result from being expressed in fuzzy conceptual terms. All the activities of a process must be focused on the result. A process must also have a customer who must be interested by the result of the process. If nobody is interested, then the process is useless. Although the customer is often a person, it could also be an organization or a department. But the customer must be able to judge if the result is correct. Finally the triggering event let us identify the beginning of the process which is not always straightforward. A process is often triggered by some action from the customer.

A process is made of a workflow of business task or activities. In the simplest case, the end of an activity initiates the next one. In more complex cases, we may have parallel tasks whose results are joined to trigger the next common activity. A task is a piece of work executed by some actor (in the UML sense) during a defined period of time. Finally, an actor is a person or, sometimes, another system.

Any task or activity may produce intermediary results or deliverables that can be used by the next tasks in the workflow. A task needs resources for its execution (people, equipments, documents, information sources, energy…). Some deliverable may also represent milestones in the process (Figure 13).

Each of the tasks is performed by some business worker (resource of the task) with the help of the information system. Then, each of the business workers will become an system actor and will use the system as described in the associated system use-case.
6. The reverse engineering process

6.1.1 Introduction

The reverse-engineering technique that we developed is based on the business tasks that the software supports. In fact, when dealing with some legacy information system, we must first know what the system is supposed to do at the business level. Then, the reengineering work is organized in three “disciplines”:

1. Specify the business scope.
2. Build the domain model.
3. Recover the architecture of the software.

Following the format of the Unified Process [Jac99] our reverse engineering process will be presented with three workflow diagrams. The first focuses on the business scope and quality attribute specification. The second deals with the reconstruction of the business process model and the associated analysis models. The third diagram will focuses on the reconstruction of the software architecture that fits the business scope. In all the diagrams, we use the UML symbol semantics as well as the UML business modeling profile. Moreover we will use the vocabulary and concepts of the Unified Process when appropriate.

6.1.2 Overview of the process

The disciplines of our process can be represented in a two dimensional picture like the RUP from which we borrowed the name of the phases. Time is represented on the horizontal axis. Each phase is subdivided in iterations. The height of the colored zone represents the amount of work performed in each discipline during each iteration. At the end of the first phase, inception, the scope of the reengineering project should be known. At the end of the second phase, elaboration, the domain model should be stable and most of elements in the scope should be reverse-engineered. At the end of the third phase, construction, the elements in the scope should be restructured according to the target architecture. Finally at the end of the transition phase, which is not covered here, the reengineered system is installed. This report will focus on the first three disciplines only. Finally, like in the RUP, the process presented here is a toolbox. Depending on the scope and size of the system to reengineer, the process could be customized: a subset only of the activities in the disciplines will be performed.

![Overall reverse engineering process](image.png)

Figure 14 Overall reverse engineering process
Each of the activities of the workflow must produce some deliverable. In the discussion below, we will represent these deliverables by the following icons:

![icon1](image1) or ![icon2](image2)

### 6.2 Business scoping

#### 6.2.1 Introduction

The first step in our process is to set the business scope and desired quality attributes of the reengineered system. Although the latter is not specific to our work, for example it is at the core of the SEI's Horseshoe model [Ber99b], the former does not seem to be explicitly dealt with in current re-engineering processes.

In today's trend for ever more business flexibility and IT cost reduction, any reengineering process should try to optimize the software assets and the IT resources. This is especially true when the reengineering option is in competition with the complete rewrite of the system in some offshore development center. Usually, the desired quality attributes of the reengineered system are defined beforehand. Then the system is restructured to reach them. But it may well be the case that an analysis of the system concludes that, because of the poor quality of the actual code, the restructuring of the system is not economically feasible. In this case the management may decide to reverse-engineer some value-added component only. In the case the system structure is so bad that no useful piece of code could be reused, the reengineering work could be limited to the extraction of the knowledge imbedded into the old system to help specify a new system. This is why the definition of the business scope is included in the reengineering iterations: the more we know the actual structure of the system, the better we can assess the economic relevance of the restructuring of the system. In other words, the goal of the reverse engineering project could change while the project develops. During each iteration, the economic relevance of the work must be reassessed to confirm or invalidate the previous goal. In summary, we could decide to reverse-engineer and restructure the whole system, to reverse-engineer only the parts of the system that must be reused or to limit the work to the extraction of the knowledge embedded in the system. These options are presented in the table below (Figure 15):

<table>
<thead>
<tr>
<th>Scope of the system restructuring</th>
<th>partial</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>component redevelopment from recovered specs</td>
<td>system redevelopment from recovered specs</td>
</tr>
<tr>
<td>partial</td>
<td>component extraction to be reused in another system</td>
<td>system redevelopment that will reuse some old components</td>
</tr>
<tr>
<td>Complete</td>
<td>System restructuring</td>
<td></td>
</tr>
</tbody>
</table>

On the other hand, if the scope of the reengineering project is changed, the desired quality attributes for the reengineered system can be impacted. For example, if the target of the reengineering is some specific component rather than the whole system, then the quality...
attributes such that flexibility, performance or testability get another scope too. Moreover, the specification of a target quality attribute may also change the scope of the restructuring task. For example, if the quality attribute is flexibility, then the examination of the structure of the code may lead to the conclusion that it cannot be enhanced without a major rewrite. In this case the option may be to restructure only the parts of the system that will be reused in a new, more flexible architecture. In the worst case nothing will be reused and the engineers will only gather the knowledge embedded in the system. Finally, depending on the difficulty to understand the actual system, the management may also decide to target the reverse engineering effort only to the parts of the system that must be reused or redeveloped. These moving reengineering targets are illustrated in the option table as shown below (Figure 16).

<table>
<thead>
<tr>
<th>Scope of the system restructuring</th>
<th>Scope of reverse engineering</th>
<th>partial</th>
<th>complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16 Changing the target of the reengineering effort

6.2.2 Workflow of the discipline

The activities represented in the next figure (Figure 18) will be included in the first few iterations, until the scope of the reengineering project stabilizes.

![Figure 17 Business scoping workflow](image)

6.2.3 Specify the business scope

In this activity we must analyze the business environment, the business trends and the alignment of the current system with the business. Then we must analyze the likely change the business may undergo in the foreseeable future\(^{52}\) as well as the competitive advantage the software provides to the company. From this analysis and the knowledge of the available software sourcing options, we must set the goal of the reengineering effort on the reengineering option table (Figure 15).

 Specification of the business scope

---

\(^{52}\) We may use framework such as the New Information Economics to perform such analysis [Ben04].
6.2.4 Specify the target quality attribute

Beyond the business process changes, another reason to reengineer a system is to improve its quality attributes. Then, at this step, we should express the target for the quality attribute of the restructured system. This depends on the business scope. For example, if the business trends ask for a more flexible business process structure, then flexibility is the key quality attribute to improve. The specification of the quality attribute should use the SEI’s scenario technique as shown in paragraph 3.3.1.

Specification of the quality attributes scenarios.

6.3 Domain modeling

6.3.1 Workflow of the discipline

The next figure (Figure 18) presents the activities to build the domain model of the system. There is a difference between the domain modeling activity of the RUP and in our reengineering process. In the latter we build the models of the concepts of the domain that let the maintenance engineer “understand” the software (i.e. build an interpretation of the software elements). This model will be a mix between a business process model and a conceptual architecture model with object responsibilities.

Figure 18 Domain modeling workflow
6.3.2 Redocument the business use case

Before diving into the software structure we should document the business process that the software supports. This will let us map the business tasks and activities to the element of the software then build the overall architecture of the system. The business use-case model represents a high level view of the business process supported by the software. Due to the level of granularity involved, we usually get a good enough picture with only one iteration.

In this activity we should only document the outline of the business process. According to [Sha01], the documentation of a process should include:

1. the process name
2. the customer of the process (the primary business actor);
3. the expected result (the result of value to the actor);
4. the triggering event
5. the other stakeholders of the process;
6. the main activities or tasks;
7. the business workers with their responsibilities and roles;
8. all the other resources to the process;
9. the time and duration of the tasks;
10. the process environment: all the other associated processes.

The first five items above can be documented through the RUP’s business use-case artifact. Specifically, the main result delivered by the software system becomes the expected result of the process and the entity for whom this result is produced becomes the primary actor of the business use-case. Moreover, rather than analyzing the business process from a business perspective, we could work backwards by identifying first the users and stakeholders of the system (the business workers). Then by abstracting the work task performed we could retrieve the sequence of actions of the associated business use-case using the business modeling discipline of the unified process.

In summary, the customer of the process becomes the primary business actor and the expected result of the system becomes the result of value of the use-case which is usually represented through its name. Finally, the triggering event and the other stakeholders are represented in the textual version of the use-case (see [Coc00]). Finally, the basic course of events and alternative course of events can be written (sequence of actions).

6.3.3 Redocument the entity model

This activity is performed while the system users are queried on their use of the system. Then each time a new business concept is used, it should be documented in a glossary and
inserted in the model of the business entities of the process. To identify the business entities we could follow the heuristics documented in the Unified Process [Jac99].

6.3.4 Build the data dictionary

After having documented the business use-case and the entity model, we must analyze the database tables and files records and build a dictionary of these records and attributes. This step only involves cataloging the data items. The meaning and usage of these data will become clear when building of the business analysis model and the system analysis model. In fact, we are well aware that a full understanding of the data model may not be possible before analyzing the use of the data by the programs. Another source of information is the entity model which plays the role of the reference information model in the conceptual data model recovery technique presented by Abbattista et al. [Abb93]. In summary, the understanding of the data will incrementally come with the building of the process model and the analysis models.

The data dictionary will record the data elements together with their type.

6.3.5 Redocument the business analysis model

From the business use-case, we can build the business the analysis model using the business modeling stereotypes of UML. Again, rather than building this model from scratch, we will start with the observation of the users of the system like we did for the business use-case (6.3.2) and identify the intermediary deliverables of the process. This work will also exploit the heuristics taken from the Unified Process [Jac99]. The granularity of the model should be such that a single worker could carry out the task.

To reconstruct the model we should work backwards from the main deliverable of the system to the first step of the process. The steps are:

1. Find the business task that produces the target business entity or result (Figure 20):
   - Identify the associated business worker.
   - Identify all the resources (business entities) consumed or exploited to fulfill the task.
• Identify the supplementary resources / deliverables (business entities) produced by the task, if any.
• Document the activities the business worker performs (Figure 21)

The object flow must show where the business entities are created and used.

2. Document, in business terms, the way the result of the task is constructed as well as its context (Figure 22):
   • Establish the components of this result (entity type, attributes).
   • Document, in broad business terms, the way the result should be computed: from what entities, with what business rules, with what intermediary results?
   • What are the important intermediary milestones / deliverables / subproducts that are generated while producing the result? Especially, is there any log file, trace file or any other security-based information generated and recorded while the task is producing its main results?

53 Often, the problem with legacy systems is that the business rules are hard coded in the software and nobody knows exactly how the result was supposed to be computed at the time of the design of the system. But a broad general idea of the way the result should be computed is generally still available in the company. For example, in a banking system, a cashier could tell that the opening of an account needs a minimum initial deposit and that there will be limits to the money that could be withdrawn each month. But the details could be forgotten.
• Find the link between the entities that are inputted to the business task and the entity that represents the result of the task. This link may be direct or indirect if some part of the result is computed.
• Try to map the resources (entities) accessed, transformed and produced by the task to the records listed in the data dictionary.

3. For all the business tasks but the last, find the mapping between the entities representing the result of some tasks and the entities used as resource by the subsequent tasks in the business process. This “implements” the dependency between tasks (green arrow in Figure 23).

4. Move one step back in the business process: focus on each entity that is used by the current task and repeat the step 1, 2 and 3 above.

To cross check the reconstructed business process model, we may rebuild the whole business activity diagram from the individual activity diagrams created for each business task. This will assure that the workflow does implement the business process (Figure 24).
Then, this diagram shows the sequence of activities performed by each business worker in the process.

![Diagram showing sequence of activities](image)

**Figure 24** Activity diagram that summarizes the process implementation

---

**6.3.6 Redocument the system use-cases**

This activity is performed in parallel with the building of the business analysis model, since the use of the system is the main source of information for the reconstruction of the business process. First, we use the classical mapping technique from business use-cases to system use case as described in the Unified Process [Jac99]. The business workers become the system actors and their use of the information system is documented through the associated system use-cases (Figure 25). Since the main source of information is the actual users of the system, they must be queried to understand the way the system is used to perform the business task they are responsible for. First, we must understand clearly what the tangible result of the use-case is (results of value to the primary actor). Then we must document the basic course of events (basic flow) that leads to the tangible result. Second, the alternative flows of events are documented. However, we should not expect to uncover all the alternative paths of the use-case from the interaction with the user because some of these paths may be triggered by environment-generated conditions. Moreover, since the use-cases are recovered from the user and not from the developers, they are unlikely to be complete. But they nonetheless represent the main part of the functionality of the system. The two steps of this activity are represented below.
1. Build the system use-cases associated to the business task (there might be several). The elements to retrieve are:
   - Primary and secondary actors
   - Results of value to the primary actor
   - Name of the use-case.
   - Triggering event
   - Other stakeholders of the use-case
   - Main flow of events.
   - Alternative flow of events.

2. Restructure the use-case model: identify subfunction use-cases [Cok00] that are reused in many use-cases. This is done by comparing the basic flows and alternative flows of event of the use-cases and extracting the common behavior (Figure 26). The comparison should not be limited to the use-cases of the same task but include all the tasks since some sub-functionality could be shared among many tasks.

It must be noted that the simplicity of Figure 25 above may be misleading. In fact, a given task may require several use-cases, depending on the number of activities of the worker.

6.3.7 Build the system analysis model

Once the models of some business tasks and the corresponding system use-cases are documented, we can build the analysis model of the system use-cases. In this activity we
use again the mapping techniques documented in the Unified Process [Jac99] and especially
the heuristics to find the analysis objects from the system use-cases. Basically, the business
entities become system entities, the interfaces to the actors (for example the screens of the
application) become the boundary objects and the responsibility for the coordination of a use-
case is represented as a control object (Figure 27). Then, a set of broad responsibilities can
be extracted from the use-cases and assigned to the analysis objects.

It is important to note that this analysis model will only represent a candidate analysis model
(first guess). In the actual source code, we could expect to find software elements that carry
out the responsibilities of the boundary objects and the entity objects in one form or another.
However, the responsibilities of the control objects will likely be scattered among many
software elements. In summary, this analysis model is used to record what we can expect to
find in the software as far as the responsibilities of the elements are concerned.

The steps of this activity are:

1. Build the candidate analysis model of each use-case.
   - Link the system entities to the business entities identified above (6.3.5).
   - Represent each screen map and I/O device with a boundary object. If the same
     screen map is used in two different use-cases, it must be represented by the same
     boundary object.
   - Assign one control object to hold the use-case specific behavior and business rules.
Record the responsibilities and collaborations of each analysis object in a CRC\textsuperscript{54} card (Figure 28).

Identify the entities manipulated by the use-case in the data dictionary.

Create a new entity object to hold the information that may be computed in the step but that is lacking in the data dictionary. Record this entity in the data dictionary.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
<tr>
<td>-------------</td>
</tr>
</tbody>
</table>

Figure 28  A CRC card to record an object’s responsibilities and collaborations

2. Synthesize and order the tasks of the business process from the analysis of each system use-case (Figure 29).
   - Check that all the results and deliverables are produced through one of the system’s use-cases.
   - Check the ordering of the business process tasks through the analysis of the production and use of each intermediary result.
   - Check the coherence of the business tasks and the system use-cases.

\textsuperscript{54} A CRC card is a way to represent the responsibilities and the collaborations of objects [Wif90]
It is important to note that we should not go any deeper into the analysis model but stay at this coarse level. In fact, this model is constructed from the analysis of the business tasks. Then, it is certain that any more detailed object structure would not match the actual implementation. On the other hand, this coarse granularity level let us represent the key responsibilities needed to implement each use case: the user interfaces and I/O devices (boundary objects) the data model (entity objects) and the use-case specific processing (control objects). This is the information we will look for in the code.

6.3.8 Build the feature cross reference

This is the last activity of the discipline. We must build a cross reference document that will point out which data structure (entity, attribute, record field, table column) participate in which business tasks and use-cases and what data structure is visible on what screen. This let us assume the link between the entity objects and the tables or file records. This document will be used in the third discipline, to help recovering the program elements that are associated to a given work task.

![Cross-referencing the data structures](image)

6.3.9 Working by iterations

This discipline is fundamentally iterative and incremental. In fact, the identification of an entity or an intermediary result leads us to identify the business task that produces it. This task uses resources or entities that are produced by other tasks which must be incrementally retrieved. For each business task, we identify the corresponding system use-cases. On each of the screens we should check that the displayed information belong to some of the entities that have been associated to the task. If this is not the case, we must look for the missing entity and the task that produces it. It is worth noting that, sometimes, the entity used by a task and displayed on a screen is produced without human intervention, by a batch subsystem for example. In this case, there is a gap in the information flow between two or more business tasks performed by humans, as shown below (Figure 31).
This is why the coherence check of the entities is very important. It assures that all the intermediary products and deliverables as well as the final result of the process can indeed be produced by the logical sequence of the business tasks. On the other hand, the identification of the attributes of the entities is difficult if we do not know the structure of the database. Consequently, we must pay a close attention to the database tables and files records before trying to assign the attributes to the entities. The following schema (Figure 32) illustrates the incremental steps of the discipline that starts from the end product (goal) of the process and works backwards until it reaches the first task of the process.

This second discipline of our reengineering method presents some similarity with the work of Gall, Klosch and Mittermeir [Gal96]. These authors try to build an object oriented representation of a procedural legacy system by building two object models. First, with the help of a domain expert, they build an abstract object model from the specification of the legacy system. Second, they reconstruct an object model of the source code, starting from
the recovered entity-relationship model to which they append dynamic services. Finally, they try to match both object models to produce the final object oriented model of the procedural system. The authors report that one of the main difficulties is the assignment of the dynamic features to the recovered objects (what they call the ambiguous service candidates). In contrast, our approach does not try to transform the legacy system into some object oriented form. Rather, it targets the building of a conceptual structure to reflect the conceptual partition of the program. This model will later be used to build an interpretation of the code. Since a high cohesion of the components (elements of the conceptual structure) cannot be assumed, our analysis rests on the dynamic aspects of the system driven by the business tasks.

6.4 Architecture recovery

6.4.1 Introduction

From the recovered conceptual structure of the legacy system we must now find the components of the information system that implement it. The problem then is to make the link between the high level analysis model and the low level software components. However we do not try to isolate clusters of program statements based on the good software engineering concepts of high encapsulation and cohesion and low coupling because:

1. We do not know if these structuring paradigms were used by the original designer of the system;
2. Even if this was the case, the maintenances undergone by the system could have damaged its initial structure.

Generally speaking, the problem is very complex and has been the source of many research works and publications. Often, in the literature, the authors try to solve the problem by designing an algorithm that will group the software elements according to some given criteria. Among the most popular techniques we find:

- Slicing: such an algorithm searches a program for all the software statements that may impact the value of a given variable [Bin96, Ver03]. For example, this technique has been proposed to extract the reusable functions from some legacy software [Lan97].
- Clustering: this kind of algorithm groups the statements of a program based on the analysis of the dependencies between the elements at the source level, as well as the analysis of the cohesion and coupling among candidate components [Mit03, Kuh05].
- Formal concept analysis: this is a data analysis technique based on a mathematical approach to group the « objects » that share some common « attributes ». Here the object and attributes can be any relevant software elements. For example, the objects can be the program functions and the attributes the variables accessed by the functions [Lin97, Sif99]. Namely, this technique has been proposed to identify the program elements associated to the visible features of the programs [Raj02, Eis03].

All these techniques try to partition the set of source code statements and program elements into subsets that will hopefully help to rebuild the architecture of the system. The key problem is to choose the relevant set of criteria with which the “natural” boundaries of components can be found. In other words, in the absence of any clue on the software structure in the
source code, what could be the similarity measurement\footnote{In other words, what is the distance metrics between elements so that we can decide what should be in a given cluster and what should not.} [Wig97] to cluster the elements? This is a traditional problem in reverse engineering and many metrics have been proposed in the literature. Wiggerts [Wig97] categorizes them in two groups:

- The metrics that measure the relationships between the entities such as number of calls, imports, inheritance relation and the like;
- The metrics that measure the intrinsic properties of the entities such as the value of some variables or attributes.

For example, in the reverse-engineering literature, the similarity metrics range from the interconnection strength of Rigi [Mul93] to the sophisticated information-theory based measurement of Andritsos [And03, And05], the information retrieval technique such as Latent Semantic Indexing (LSI) [Mar04, Kuh05] or the kind of variables accessed in formal concept analysis [Sif99, Ton01]. Then, based on such a similarity metrics, an algorithm will decide what software element should be part of the same cluster [Mit02]. As a simple example, Rigi clusters software elements whose their interconnection strength exceeds some predefined threshold.

In contrast with these techniques, our approach is “meaning-driven” i.e. we clusters the software elements according to the supported the business tasks and functions.
6.4.2 Workflow of the third discipline: Architecture recovery

- Assess the current quality of the system
- Assess the architecture
- Make re-architecture proposal
- Run the use-cases and record execution traces
- Map the traced functions to the visible high level structure of the code
- Identify the target work task & corresponding use-cases
- Redoc the visible high level structure of the code
- Extract knowledge from the code
- Rebuild a meaningful architecture
- Assess the architecture
- Make re-architecture proposal
- Generate the call graph for the traced functions.
- Bottom-up validation of the analysis model
- [quality improvement]
- [component extraction]
- [knowledge extraction only]
- [late iterations]

Figure 33 Architecture recovery workflow
6.4.3 Assess the current quality of the system

This activity is required when the goal of the reengineering project is the improvement of the quality of the system. The assessment technique is based on the quality scenarios defined in the first discipline (§6.2.4) and presented in § 3.3.1 [Cle02]. In the case of maintainability we could also explore the maintenance database of the company.

This assessment must be documented in a software assessment document which will contain the quality attribute scenarios together with the result of its execution on actual system.

Quality attribute assessment document

6.4.4 Redocument the visible high level structure of the system

As a first step of the software structure reconstruction we must document of the visible high level structure of the system. This is represented by the containment relationship between the directories, libraries, files, classes and packages. Then, a model of this structure must be reconstructed. In the figure below (Figure 34) we present a model where a part of the system would be written in a procedural language and the other part in an object oriented paradigm. This model should not go deeper than the level of files or classes. Following the work of Krikihar [Kri99] the visible high level structure could also be represented by an algebraic formula using the part-of relationship: part-of(x,y) is true if artifact x is included in artifact y. However a visual representation is easier to work with.

The information provided by this model will allow us to represent the dependencies between the software artifacts due to the implementation of the business task.

Figure 34  System and packages containment hierarchy

56 Which belongs to the development view in the 4+1 model of Kruchten [Kru95].
6.4.5 Identify the target work task & corresponding use-cases

Depending on the scope of the reengineering project (§6.2.3), we may focus on a subset of the full system only. For example, we could be interested to extract the components implementing a given business function. Moreover, depending on the actual structure of the legacy system, the scope of the project may narrow during the process if a full restructuring appears economically irrelevant (§6.2.1).

In this activity we should first identify the business tasks that correspond to the reengineering scope. On the other hand, if the target of project is the enhancement of the quality of the software, we should identify the subsystem to be improved. For example, if maintainability is the quality attribute to improve, then we could analyze the maintenance database of the company to identify the subsystem that is hardest to maintain. On the other hand, if the problem is with the usability of the software we must find the subset of the system that is harder to deal with than the rest. Finally if the performance of the system is a concern, we should identify the bottlenecks among the subsystem. Once the business task is known, we must select the associated system use-case.

On the other hand, according to § 3.2, the reengineering project could also be driven by a profound change in the business process due to some global trend in the business. Then, the impact of the change must be analyzed on the business process model and the corresponding business tasks identified (top layer in Figure 10). This will represent the scope of the reengineering to be done.

6.4.6 Run the use-cases and record execution traces

In order to identify the code that implements a given business function we work both top down and bottom up. In the top down part, we record the execution trace of the software\(^{57}\) by running all the use-cases associated to the business task to investigate. First we must run the basic flow of the use-cases. Then the execution traces are recorded together with the name and input parameters of the corresponding use-cases (Figure 35). Second, we should run the alternative flow of the use-cases. However, as said before (§6.3.6), since the use-cases are recovered from the user and not from the developers, they are unlikely to be complete. Especially we should not expect the alternative flow of events be exhaustive. Then the execution trace will not represent all the code associated to a given business function.

In our methodology, we are looking for the code that implements some business task because the business model is the domain model with which we build an interpretation of the code. We do not want to target a single feature as proposed in the work of [Won99, Eis03]. It is worth noting that the use-cases play here the same role as the test cases in the execution slicing approach of Wong et al. [Won99]. However, in our work, the test cases are not arbitrary but represent actual business-related function of the system. Of course, we cannot run the use-case with all possible input values. Rather, we must restrict ourselves to the typical values.

\(^{57}\) Depending on the implementation language and programming environment at hand, the execution trace could be recorded with the tools of the environment. If this is not possible, we must insert tracing function in the source code and recompile it. This insertion could be done with the help of some instrumentation program or, in the worst case, by hand.
Since the sub-function use-cases ("included" use-case) are reused by several primary use-cases, they will lead to shared code in the execution trace. To ease the identification of this shared code we could run the sub-function use-case separately. In summary, each run of the primary use-cases and sub-function use-cases will correspond to an execution trace that must be recorded together with the name and input parameters of the use-case.

Table of the use-cases and associated execution trace

6.4.7 Map the traced functions to the visible high level structure of the code

Once the execution traces are recorded, we must map the function in the execution trace to the visible high level structure of the code. Basically, all the functions must be part of some class or file that has been identified before. This mapping will let us identify the elements of the visible high-level structure that implement the use-cases of a given business task, as shown in the figure below. The executed code is then the one that implements the business functions. In non object-oriented software the classes in this figure would be replaced by files, although the granularity would be different (there are usually much less files than classes in a non object oriented system with the same number of NCLOC). This work is done for all the executed use-cases.

---

58 If the execution trace is obtained with a debugger, the name of the class/file to which a given function belongs is often displayed. If this is not the case, the source code may have to be analyzed to identify the class/file to which the function belongs.

59 NCLOC: non-comment lines of code
This technique lets us discover the coupling between the visible high level elements involved in the implementation of a given use-case (elements colored yellow). It must be noted however that the number of directories / packages in the figure is somewhat misleading. In fact, a large information system may have hundreds if not thousands of packages and/or directories. The identification of the files or classes together with the packages and directories that map to some use-case already represents a valuable result along the way to recover the meaningful structure of the code. This mapping must be recorded in a document.

These high-level elements belong to the development view of the code that implement a given business task. All these classes (or files) are then somewhat coupled. This is also true for their containing packages and subsystems. In the next activity we will extend this set of classes (or files) with a static analysis of the code.

6.4.8 Generate the call graph for the traced functions

Once the execution traces are recorded, we must complement the set of function associated to each use-case by performing a static analysis of the code to build a context-insensitive call graph [Gro01]. Such a call graph is a pair \((N,R)\) where the nodes in the set \(N\) are functions and \(R\) is the “call” relation i.e. there is an edge between the functions \(f_1\) and \(f_2\) if the implementation of \(f_1\) includes a call to \(f_2\) regardless of the conditions of such a call. Then, any function that is called from one of the alternative of a conditional statement or from the body of a loop statement must be considered an edge in this graph. Again, the use-case’s flows of events retrieved from the interaction with the users are unlikely to be complete (6.4.6). Consequently an execution trace of such a use-case will not execute all the functions that

---

60 A special difficulty is posed by the so called “dynamic calls” i.e. when the function call is not hard coded in the source code but dynamically constructed at run time. This is represented by the perform: message in the Smalltalk language, the invoke() method of the Java reflection mechanism or the CALL USING instruction of the Cobol language. There is hardly another way to get around this problem than analyzing the code by hand. However, the use of such a dynamic call is rare.
actually implement the complete use-case. To complement them, we must transitively find all the functions that are called by the traced functions. In the figure below (Figure 37) we show an example of such a call graph. The functions in red represent the traced functions. The set of functions to retrieve, that we call the extended set of function, are surrounded by the dotted line. These are the functions that could potentially be executed when running all the possible scenarios of the use-case\(^{61}\). The extra function will include variants of the behavior triggered by the initial use-case execution.

Then the extra functions must be mapped to the visible high level structure of the code, like we did for the traced functions (§6.4.7). The additional high level elements involved (classes, files, packages and directories) are then coupled to the other high level elements (such as Figure 36). This work will complement the report on use-case mapping to the visible high level structure.

---

\(^{61}\) In general, though, we do not know if there exists a configuration of use-case inputs that will actually execute the extra function. In fact, since most of the languages in which the legacy programs are written have the power of the Turing machine the problem, for such languages, is known to be undecidable [Hop79]. This is why the extended set of function is a superset of the set of function that are executable when running the use-case.
However, it must be noted that the object oriented languages pose some special problems when trying to recover the call graph because of polymorphism and inheritance relationships. In fact, if the declared type of the receiver object in the calling function (i.e. the object that will execute the called function) is the root of an inheritance tree or if it is an interface, then we must record all the possible classes for the receiver object (Figure 38). In this case a given function call may lead to a set of edges representing all the possible functions like in a conditional statement (Figure 39).

In the case of an object oriented code, it is worth noting that, although the call graph may be huge, it is a subgraph of the graph retrieved by following the association relationships between the classes. In fact, even if some association exists between two classes (i.e. a variable is typed with a class), it might be the case that the flow graph of a given use-case does not use this association.

6.4.9 Bottom-up validation of the analysis model

After the top down identification of the code from the running use-case and call graph, we must work bottom up from the extended set of functions (6.4.8) of the use-case. This step will let us validate our assumptions about the entity and the boundary objects associated to the use-case (6.3.7) as well as the cross reference to the data structures (6.3.8).

First, the source code of the extended set of functions is searched for any table or file access. This will represent the data structures that are actually (or potentially) accessed during the use-case execution. Once such a table or file is found, we must compare it to what was expected in the analysis model (Figure 40). Then, this model as well as the cross reference document (6.3.8) must be corrected accordingly. Depending on the legacy system to analyze, the tables or files may be declared both outside of the program code and/or directly within it. The technique to retrieve this information is highly system specific but it should normally not be too hard as each language has I/O specific statement to be searched in the code.

62 For example in a Cobol batch program running under IBM MVS, the files accessed are declared in the job control language (JCL) files and the record structure in the data division section of the program.
Figure 40 Validating the entities in the analysis model

Then, the mapping from the extended set of functions to the high level structure of the code lets us find the classes (files) to which the entity objects could be associated (Figure 41). The inference is the following:

1. The access to the tables or files by the functions is retrieved in their source code;
2. The functions are mapped to the classes (or program files) (§6.4.7);
3. The mapping from the tables to the entity objects is recorded in the cross reference document (§6.4.7);
4. Then, we could assume that some of these classes (or program files) implement the entity objects.

Once the validation of the analysis model for a use-case is done, we must double-check the chain of dependencies between the analysis models associated to all the use-cases of the target business tasks (Figure 29). For example, if a given table is not accessed by the extended set of functions of a use-case then the corresponding entity object in its analysis model must be challenged. If this entity is removed, we must rework the dependency chain between the individual analysis models so that the whole model stays coherent.
example, we may have to find another object to fill the gap between two analysis models (Figure 42). On the contrary, if a new entity object is associated to an analysis model, we must find its purpose and connections to the other analysis models.

Finally, we must keep in mind that some of the files or tables may not be linked to any specific business task. This is the case, for example, with parameter or logging files. Then, these files are likely to be accessed by the code associated to several use-cases.

Second, the source code of the extended set of functions is searched for any screen-related functions. These are highly specific to the programming environment and programming language considered. Normally, these functions are well documented in the programming manuals and their identification should not be too difficult. In an object-oriented environment, there often exist predefined classes that implement the windowing mechanisms. In such a case the result of the mapping from the extended set of functions to the classes (§6.4.7) is used to identify the classes that belong to the windowing system (Figure 43). In a non object-oriented programming language, the classes in this figure would be replaced by files. The inference is the following:

1. the function are mapped to the classes (or program files) (§6.4.7);
2. the reference documentation of the programming environment let us know what are the display classes and functions;
3. the display classes (or program files) are identified in the set of classes (or program files) associated to the use-case;
4. then, we could assume that these classes implement the boundary objects.

However, we must keep in mind that in a badly designed object-oriented system, a given display-related class may also hold business logic code and even data access code. On the other hand, this will more often be the case for program files in non object-oriented legacy
systems as the clean separation between views and models is a relatively modern concept. This must be sorted out.

At the end of this activity, both the system analysis model and the data structure cross reference documents must be updated to reflect the changes that have been made during this bottom-up validation.

Finally, this work on the mapping of the boundary and entity objects to the classes or program files can be summarized on a model (Figure 44). However, the control objects cannot be mapped to a single class or file as their responsibilities will likely be scattered among many programming elements. This can be seen in the summary model where the mapping between the control object and the classes could be assumed to be the difference between the set of classes associated to the use-case (Figure 36) and the set of classes already mapped to the boundary and control objects.

This is a reasonable hypothesis only if the use-case does not include any sub-function use-cases (Figure 26). If it does, the difference may include the implementation of the sub-function use-case. This is also why it is important to model all the shared sub-function use-case to help the interpretation of the code elements.

---

63 It was popularized in the early 1980’s with the MVC concept of Smalltalk. But it actually took a long time before this concept was taught in programming courses.
In the next activity we will investigate the inner structure of the directories and packages to extract the use-case specific code.

### 6.4.10 Rebuild a meaningful architecture

**Introduction**

In this activity we must sort-out the code that is specific to each use-case and rebuild the corresponding high level architecture of the code. First, we must work at the class and file level. Then we will go deeper into the code. Let us have the set UC of use-cases that have been reverse-engineered UC= {UC₁…UCₙ} and let us define the following two functions:

- **classes(UCᵢ)** : returns the set of classes associated to the use-case UCᵢ. These are the classes that contain the extended set of functions for the use-case.
- **specific(UCᵢ)** : returns the set of classes that are specific to the use-case UCᵢ i.e. that contain the functions that belong only to the extended set of functions of this use-case.
- **common({UC₁…UCₖ})** : returns the set of classes that are common to the set of use-cases {UC₁…UCₖ}.

Then we have:

\[
\text{specific}(\text{UC}_i) = \text{classes}(\text{UC}_i) \setminus \bigcup_{j \neq i}^{\text{classes}(\text{UC}_j)}
\]

The set of common classes for the set of use-cases is simply defined as:
\[ \text{common}(\text{UC}) = \bigcap_{\text{UC}_i \in \text{UC}} \text{classes}(\text{UC}_i) \]

In non object-oriented systems we have:

\[ \text{files}(\text{UC}_i) : \text{returns the set of files associated to the use-case UC}_i. \text{ These are the files that contain the extended set of functions for the use-case.} \]

\[ \text{specific}(\text{UC}_i) : \text{returns the set of files that are specific to the use-case UC}_i \text{ i.e. that contain the functions that belong only to the extended set of functions of this use-case.} \]

\[ \text{common}([\text{UC}_1 \ldots \text{UC}_k]) : \text{returns the set of files that are common to the set of use-cases \{UC}_1 \ldots \text{UC}_k\}. \]

Then we have:

\[ \text{specific}(\text{UC}_i) = \text{files}(\text{UC}_i) \setminus \bigcup_{\text{UC}_j \in \text{UC}} \text{files}(\text{UC}_j) \]

The set of common files for the set of use-cases is defined the same way as:

\[ \text{common} (\text{UC}) = \bigcap_{\text{UC}_i \in \text{UC}} \text{files}(\text{UC}_i) \]

This let us draw a map of the high level elements involved in a set of use-case. In the following figure (Figure 45), we represent both the elements that are unique to a given use-case (same color as the use-case) and the common elements (green).

The common classes (files) between the two use-cases (common([UC}_1,\text{UC}_2)) could represent either the common entity and boundary objects in both use-cases, the implementation of functions common to several analysis objects or the objects of a common sub-function use-case. In the following figure (Figure 46), the object colored green represent shared analysis objects which are not associated to the sub-function use-case. In the case where a sub-function use-case is included in both use-cases, we must first identify the classes that belong to it. Then we could identify, in the remaining set of objects, the classes that could be attributed to the common analysis objects of the primary use-case.
Finally it must be noted that, sometimes, an analysis object that is common to two use-cases may not correspond to a common class (or file) in the code. This is the case where the code is duplicated in both use-cases. On the other hand, some code (class, file) may be omnipresent (i.e. code that is executed for most of the use-cases and that does not represent a common business function). This could often be attributed to some aspect of the program.

Recovering a meaningful high level architecture of the code

When all the use-cases that belong to the target business task are processed we must group the classes (files) according to the use-case they implement and record the possible mapping to the analysis objects. This step must first start from the sub-function use-cases then proceed with the primary use-cases. Once this is done for all the use-cases, we must analyze the visible high level structure to see if some logical grouping of the classes (files) emerges. Among the possible grouping criteria we have:

- Grouping by analysis object type (example: the classes that implement the boundaries);
- Grouping by use-case (example: the classes that implement a given use);
- Grouping by information source (example: the classes that access a given database);
- Grouping by actor (example: the classes that implement the interaction with some user role).

Moreover a structure could be based simultaneously on several criteria. As a result, we can sketch the high-level structure of the code according to the discovered grouping. For example, the Figure 47 represents a grouping by use-case.
The next figure (Figure 48) adds a level of detail to the recovered architecture by showing the correspondence with the analysis objects. However, even if some logical grouping was done when the program was initially developed, it might well be the case that the maintenances did not respect it [Ton05]. In this case, some apparent grouping may contain classes that do not respect the criterion.

On the other hand, if the visible high level structure is very shallow then we must choose a grouping of the software elements that will optimize the mapping to the domain concepts. At the same time we will try to optimize loose coupling and high cohesion. But we insist that the chosen grouping must lead to a good mapping to the domain concepts and should not be driven by the search for loose coupling and high cohesion. In this case, the recovered architecture will be a virtual structure that does not correspond to any tangible software construct.
The recovered architecture must be recorded in an architecture document that will be complemented as the understanding of the structure become clearer.

Architecture document

Going one level deeper

In the work above, we identified the classes according to their involvement in the execution of a set of use-cases. However, even if a class is shared among several use-cases, there might be methods specific to each use-case. Then, the next step is to compare the call graphs of the use-cases and to map common code to common behavior. The work is more difficult as there might be hundreds of function calls for each use-case. Since the granularity of the visible high level structure of the code is smaller for object oriented systems (classes) than it is for non object oriented systems (files), this code level analysis is even more important for the latter.

This step is highly iterative and starts with the close examination of the CRC cards built during the reverse-design of the system analysis model (6.3.7). The basic idea is to map the common behavior represented in the CRC cards to the common parts of the call graphs. Of course this technique is very dependent on the fact that the common behavior among use-cases is represented by some common code. In particular, if the common code is duplicated then this technique will not provide substantial results.
Moreover this technique will allow us double-check the mapping between some of the analysis objects and the classes (files) in the code. Here is how to proceed (Figure 49):

1. Identify the use-cases whose common behavior is not attributable to common analysis objects.
2. Gather the CRC cards of the objects of these use-cases.
3. Gather the call graph for these use-cases.
4. Identify the common subgraphs in the call graphs that cannot be attributable to the common analysis objects.
5. Identify the common responsibilities in the CRC cards of the analysis objects.
6. Try to map the subgraph to the common responsibilities.
7. Check if the functions in the common subgraphs belong to the same classes (files).
8. Check that the classes (files) are associated to the same analysis object type.
9. Try to reduce the discrepancy between the responsibilities attributable to the common subgraphs (i.e. try to see if the responsibilities, although expressed differently, could correspond to the same behavior) until a reasonable mapping emerges.

Once all the reasonable mapping is done, we should update the architecture document with the link between the analysis objects, the responsibilities, the functions of the subgraph and the classes (files) in the code.

Figure 49  Mapping common responsibilities to common subgraphs in the call graphs
Finally, the common function can be mapped to the common classes to reach the lowest architecture reconstruction level (Figure 50).

In the case of an object oriented program, it could be surprising that we do not use the generalization relationships between the classes as a criterion to recover the architecture. This is because our technique does not rest on syntactic elements to group the classes but on the link to the domain model (i.e. the semantics).

We believe our call graph reconstruction technique is efficient because it represents the actual use of the classes in the implementation of a business task. In the case where the generalization links are important, they will automatically be taken into account through the reconstruction of the call graph (Figure 38 and Figure 39). When several use-cases are analyzed as presented above, a summary of the recovered architecture can be represented at different granularity levels together with the corresponding domain concepts. This the essence of a meaningful architecture. In the next figure (Figure 51), we present the levels of granularity resulting from our reverse-engineering process.
The next figure (Figure 52) shows the dependencies between the actual software elements at different granularity levels. The arrows represent the result of the mapping from the use-cases to the classes as well as the part-of relationship between the classes and their enclosing packages and subsystems. Since the classes hence the packages and subsystem could be shared among several use-case and business tasks, we can represent the dependencies at each level in the hierarchy. Then, this figure shows the smooth transition from the business process structure to the software structure.
Figure 52  Transition from the business process to the structure of the code

The result of all the architecture reconstruction work must be recorded in the architecture document.
6.4.11 Assess the architecture

The last activity in the architecture reconstruction is to assess the suitability of this architecture to the goal of the reengineering project (component reuse, quality enhancement, business process change). Once the architecture of the legacy system has been recovered, there are many options depending on the business conditions. We could restructure the system, port it to a modern platform, reuse some of the components or extract the business knowledge from the old system. This decision goes well beyond the technical level to reach the business strategy level which must take into account the business context. In particular, the future shape of the business process as well as its required flexibility is important. But to enable this flexibility, the underlying software system must support it. If this is the case, a “simple” porting of the legacy system to a new development platform may be enough. On the contrary, if the reengineering must increase the flexibility of the software, then the analysis of the business processes and their possible evolution is mandatory. More globally, the desired qualities (non functional attribute [Bas03]) of the software system are the main drivers behind a reengineering strategy. But, depending on the current structure, the corresponding re-structuring effort may be too important with respect to the cost of a complete rewrite. In this case, the reverse-engineering result will be completed by a knowledge extraction process to generate the specification of the new system. However, in this case, the level of detail to go into is much finer.

6.4.12 Make re-architecture proposal

If the goal of the reengineering process is to improve some quality attribute of the software then we should run the quality attribute scenario (6.2.4) to determine the suitability of the recovered architecture to the targeted quality attribute. However, the amount of effort needed to restructure the code depends on the coupling of the software elements that belong to different domain concepts at the same level of granularity. This can be partly measured through the metrics presented in the introduction of §6.4.10.

For example, if specific(UCi) / classes(UCi) is << 1, then we may have a lot of difficulty restructuring the code that implements UCi without impacting the code that implements the other use-cases. On the other hand, even if this ratio is close to 1, the classes (or files) that implement a use-case may be located in the same package (directory) as all the other use-cases. In this case the restructuring of the package may have to be done. If we must retrieve the code that implements some high level business function so that it could be packaged as a reusable component, the problem is to decide on the extent of the code to include in the component. Often, the visible boundaries of the software structure (file, package, directories, subprogram and program) will not match the functional boundary of the component to extract. Then the problem is to find an optimum between size of the code to be extracted and the amount of the restructuring work. As an example, the interface slicing technique of Beck and Eichman [Bec93] targets the extraction of the code that implements some specific function from a given component based on its interface definition. At the other extreme, we could decide to include all the packages and subsystems that contain at least one function to be reused. In short the extraction technique must optimize semantic coverage (the purpose of the function) while minimizing the code to alter.

6.4.13 Extract knowledge from the code

This is the activity to perform if the legacy system is unsuitable to restructuring. In this case, we should extract the business rules from the code so that a specification for the new system could be written. In this case, the code should be analyzed by hand to the level of the single algorithms. The associated techniques go beyond the present report and will not be dealt with any further.
7. Conclusion

This work started with a large survey of the current reverse-engineering and reverse-architecting techniques. From this preliminary work we have drawn the conclusion that the recovery of a software architecture from the source code cannot be separated from the higher level goal which is to understand the software. In fact, when a legacy system must be maintained or restructured, we need to understand the purpose of the elements of the code so that the right restructuring decisions can be made.

In the literature, many techniques have been proposed to recover the software architecture of legacy systems based on sophisticated syntactic techniques. But the problem with these techniques is that the recovered architecture (i.e. a set of software clusters or components together with their relationship) is not guaranteed to map to any meaningful concept in the application domain. On the other hand it is well known that a fair part of the cost of maintenance is related to the understanding the code. As the program must perform some useful business function, it is compulsory to understand the role of the code elements with respect to the business domain.

In a recent text book, Tonella wrote “most information about a program is in the source code” [Ton05 p.170]. This is trivially true if this statement refers to the syntactical level only. However, we believe this to be wrong at the semantic level: there is no way to draw the meaning of a program, which lies outside the program, from the program only. We think that the software, like any other language, expresses a limited set of facts about some problem domain. These facts get their meaning in the context of the problem domain only.

Then we proposed the term “meaningful software architecture” to point out a structure that maps well to the concepts of the application domain. From this definition we proposed a reverse-engineering process to recover a meaningful architecture from the source code of a legacy information system. This process is adapted from the Unified Process but works both top down and bottom-up. It is “integrated” in the sense of [May95]. Then some of the main UML software models can be recovered. From these models the restructuring work can be performed to target the desired quality attributes for the system, specified through some quality scenarios.

Following the software comprehension framework of N. Gold [Go100], the knowledge base of the human “processor” is, in our work, the business process model augmented with the software analysis model. This choice has been made for the following reasons:

1. The business model is the level at which some of the major changes of the software are specified (§3.2). In other words, this is the level at which the users often think when asking for a change in the software.
2. It is the most reliable model we can rebuild when the only source of information beside the source code are the actual users of the system.

On the surface, our work may appear close to the one of Eisenbarth and Koschke [Eis03]. However the major differences are:

1. The scenarios we use have a strong business-related meaning rather than being built only to exhibit some features. They are actual instances of use-cases.
2. The software components we seek must be interpretable in the domain model. They are not arbitrary.

---

64 In the sense of the UP [Jac99]
3. The whole process aims to rebuild an architecture that maps to a domain model. We believe this to be key to program comprehension.

It is worth noting the central role played by the notion of scenario in this work. First we must re-document the use-case of the system in order to recover the business process model that is supported by the software. Next, a subset of the use-case is used as test case to gather the execution trace of the system. Finally the quality attribute to improve are expressed as quality scenarios and are used to drive the re-structuring of the system.

Our future work will target the design of a reverse-engineering environment to help the software engineer make the mapping between the domain model and the software structure at different granularity levels. We envision a system where the engineer could build the business process model as well as the other UML models and link them to the software elements. Then, the properties of the domain model could be used to infer the facts to be searched for in the software.

As a final word, we would like to thank the “Reserve Strategique” of the Swiss Confederation for the support of this research (project number ISNet-62).
8. References


Springer 1999.


[Mul90] Muller A.H., Uhl J.S. – Composing Subsystem Structures using (k,2)-partite Graphs. IEEE ICSM’90


[Riv02] Riva C., Rodriguez J.V. – Combining Static and Dynamic Views for Architecture Reconstruction. Proc IEEE CSMR’02.


