Author

Dr. Juergen Rilling

State of the Art Report on Component Substitution

Dr. Juergen Rilling
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC H3G 1M8

State of the Art Report on Component Substitution

Approved for release by

Dr. Juergen Rilling

State of the Art Report: Software Component Substitution W7701-5-2936

Abstract

The problem of managing the evolution of complex and large software systems is well known. Organizations are under tremendous pressure to evolve their existing systems to better respond to marketplace needs and rapidly changing technologies. This constant pressure to evolve these systems is driven by every growing expectations of the customer for new enterprise standards, new products and system features, and improved performance. Evolution is therefore required to cope with new software releases and manage hardware and to avoid software obsolescence.

Component substitution can be seen as a specific instance of a software maintenance activity. It typically involves the comprehension of an existing system, the identification of the scope of the modification, conflict identification, component adaptation, and implementation of connections between components. The challenges for a maintainer start with the identification of the scope of a component. This is particularly true in the context of a large system and/or a system the maintainer is not familiar with

This report on component substitution provides a state-of-the-art survey of (1) techniques that can support component substitution at various abstraction levels, (2) their information needs and the (3) specific information these techniques provide. The report also provides an initial survey of tools supporting these techniques and provides an overview of current maintenance models that can be applied to support a substitution process. The report concludes with some critical evaluation and recommendations relevant for the development of a new component substitution process.

This page intentionally left blank.

Executive Summary

Component substitution can be seen as a specific instance of such a software maintenance activity. The questions and challenges that arise are manifold, from how to identify a component in such a large system, what are the services provided by a specific component and how is the component dependent on other components or how might it affect the overall system behaviour. Other issues that may arise during the component substitution itself are that the new component might not match exactly the system requirements or the specifications of the component to be replaced. Component substitution is a multi-dimensional problem domain, where maintainers have to deal with challenges that include technical aspects, like the abstraction level, the type of component, the type of substitution and the environment in which the substitution is being performed. Maintainers will have also to deal with incomplete, inconsistent information resources, e.g. documentation versus source code. Furthermore, the process is also affected by other aspects that are not directly related to the component, e.g. maintainer's expertise and understanding of the system, the availability of tools and techniques to support both the acquisition and interpretation of knowledge relevant to the substitution process, as well as support for performing and validating the substitution. The focus of this state of the art report on component substitution is on surveying techniques and tools that can support a component substitution process, as well as their information needs and the information they provide towards such a substitution process.

The report provides a state of the art review of techniques, their information needs and the information relevant to component substitution. The review of these techniques is categorized into different abstraction levels, based on the granularity of the component to be substituted: Statement level, corresponding to components at statement or equivalence level, function or class level, feature/package level and subsystem/system level. Furthermore, each of these categories is further divided into COTS and FOSS components, due the fact that the availability of techniques supporting these types of components differ significantly. The review also discusses in a separate section techniques and issues related to supplementary techniques and artifacts that can support a component substitution, like document analysis, knowledge discovery and traceability among other artefacts.

An initial survey of tools supporting these techniques relevant to component substitution at the various abstraction levels is provided. The tool survey main focus is on tools supporting a component substitution process and excludes more general comprehension and maintenance tools. The report also presents existing process models supporting either software maintenance or some aspects of component substitution. This review is followed by a brief discussion on existing challenges and open issues related to developing a component substitution process. The report concludes with an overview of the surveyed techniques, some recommendations on the development of a component substitution specific process and conclusions.

One of the conclusions of this report is that any attempt to derive a full-automatic comprehension substitution process based solely on reverse engineering is poised to fail. This is due to the fact that it is impossible to recover all the domain expertise and functional and non-functional requirements from existing source or bytecode that influenced the original design or implementation. Both, program comprehension and component substitution will have to take into consideration the current understanding a maintainer has of the system under investigation, it will have to support the acquisition of new and additional knowledge, as well as facilitating some type of integration of knowledge, information needs, tools, techniques and information provided. Program models or better component substitution models will have to reflect these issues by providing a representation that will closely match both a user's mental model of the system as well as the ability to reason about knowledge in these models. The process support has to go beyond simple querying and browsing. Rather it will require some inferring of knowledge to provide more meaningful abstractions that support the substitution process.

Revision History

Version 1.3 (14-11-2006) Major revision of grammar, style, references are updated
Version 1.2 (16-5-2006) Updated references, included citation index, provided a citation view of the references
Version 1.1. (25-4-2006): Complete grammar, style revisions, critical views are included, abstract revised, tools section extended, references reorganized
Version 1.0 (29-3-2006) Complete report submitted

Release plan:
Version 1.3. (30-5-2006): Incorperate feedback from meeting and add some additional focus with respect to particular type of components.

Table of contents

List of Figures

List of Tables

1 Introduction

The challenge in managing the evolution of complex and large software systems is well known. Increasingly, organizations view their software assets as investments that grow in value rather than liabilities whose value depreciates over time. At the same time, these organizations are under a tremendous pressure to evolve their existing systems to better respond to marketplace needs and rapidly changing technologies. Evolution is therefore required to cope with endless new software releases and manage hardware and software obsolescence.

The repeated modification of a legacy system has a cumulative effect that often leads to an increase of a system complexity. Eventually, existing systems become at the same time too fragile to modify but also too important to discard; organizations must therefore consider modernizing these legacy systems so that they can remain viable. Reengineering is typically used an approach to transform a legacy system into one that can evolve in a disciplined manner. To be successful, reengineering requires insights from different stakeholders and artefacts, e.g. software, managerial and economic perspectives. Failing to integrate these stakeholders in the reengineering activity; will lead to products that unsatisfied and frustrated because users may not see the benefit of these initiatives.

Most of the existing research in the software maintenance domain is focused on the evolution of legacy systems based on legacy technologies. However, there is a clear need to address also the evolution of systems that are based on evolving technologies, like component-based systems. Supporting the evolution of these newer technologies raise various new challenges and issues about software comprehension and evolution, typically not found in traditional legacy applications. Component substitution corresponds to a specific evolution task found in modern systems. In this context component substitution corresponds to the replacement of existing component in a given system with a component that meets the new system requirements. Component substitution involves more than typical program comprehension and code level analysis. The substitution of software components affects quality attributes, which are not directly documented in an application's source code. Although software component products are attempting to simulate the "plug-and-play" capability of the hardware world, in today's reality, software components seldom plug into anything easily. Most products require some amount of adaptation to work harmoniously with the other commercial or custom components in the system. The typical solution is to adapt each component through the use of "wrappers," "bridges," or other "glueware." It is important to note that adaptation does *not* imply modification of the component. However, adaptation can be a complex activity that requires technical expertise at the detailed system and specific component levels, as well as at least a partial understanding of the system and the components to be modified/substituted.

Before a component can be replaced or substituted with a new component, reconstruction of an existing component has to take place, requiring the reconstruction of a model to incorporate architecturally significant concepts. These concepts include for example, architectural level components and connectors including design patterns, component distribution information and architectural decisions. These types of information can typically not be directly extracted from source code without the domain knowledge of software engineers/architects. It must be constructed and expressed during the reconstruction process by producing models at different abstraction levels. More importantly, the mapping between these different abstraction levels must be retained and available during the component comprehension and substitution process. However, the required levels of abstraction can not be generalized due to the diversity of the software artefacts and the goals of the substitution being performed.

1.1 Scope and Focus of the Report

Component substitution typically involves the comprehension of an existing system, identification of the scope of the modification, conflict identification, component adaptation, and implementation of connections between components. Usually, it is not easy for a maintainer to identify the scope of a component in the context of a large system. The questions and challenges that arise are manifold, from how to identify a component in a large system, what are the services provided by a specific component and how is the component dependent on other components or affect the overall system behaviour. Other issues that arise during the component substitution itself are that the new component might not match exactly the system requirements or the specifications of the component to be replaced. Component substitution therefore has to support the identification and comprehension of components and resolve any conflicts (or mismatches) between an existing component (the component to be replaced) and the requirements that the new component (substitute) has to match. The main focus of this state of the art report is to survey techniques and tools that can support a component substitution process, as well as their information needs and the information they provide towards such a substitution process.

1.1.1 Definitions

Software components are real and are already profoundly altering the practice of software engineering. Despite signs of progress, the challenges in evolving systems that are constructed predominantly from *commercial off-the-shelf components* have yet to be addressed adequately in both industry and research.

It has to be noted that both the term and what constitutes a component is very ambiguous and a large number of different definitions and interpretations of the term component can be found in the literature. A simple and compact definition is the following: "binary units of independent production, acquisition and deployment" [Szy98]. But also looser definitions can be found: "a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files" [MOR02]. Yet another definition of a component is that it corresponds to "a unit of design (at any level), for which a structure is defined, a name identifying the component is associated, and for which design guidelines, in the form of design documentation, are provided in order to support the reuse of the component and to illustrate the context where it can be reused" [MOR02].

Components can in general be classified in two main categories.

Conceptual component: A model/schema (or a subset of) to be reused; following an approach based on object modeling, it may be specified with the Unified Modeling Language (UML); other more specific models may be adopted in other application areas, such as for workflow-based components.

Software component: A coherent software package that can be independently developed and delivered, has explicit and well-specified interfaces for the services it provides and for the services it expects from the others, can be composed with other components, perhaps customizing some of its properties, without modifying the component itself. The term *component instance* is used to distinguish the specification of a component and the executable that implements that specification from a particular installation of that executable and a "running" incarnation of that executable. In an object-oriented approach, a component is a set of classes assembled together to be deployed as a single software unit; the component instance is the object (set of objects) which is the runtime manifestation of a component when composed within a particular application.

In the context of this report we adopt the following general component definition:

Definition:

A *component* is a software component that consists of set of related entities that together have either functional or abstract cohesion. They therefore form a *cluster* that corresponds to a set of subprograms (functions and procedures), variables, constants, and user-defined types proposed as a candidate component.

Based on this definition a *component* therefore corresponds to a group of related elements with a unifying common goal or concept relevant at the architectural level. An *atomic component* is a *non-hierarchical* component that consists of related global constants, variables, subprograms, and/or user-defined types. As opposed to an atomic component, a *subsystem* is a *hierarchical* component consisting of related atomic components and/or lower-level subsystems.

Each *existing* component must have an implementation that may be in a variety of forms: executable, source code/file, object code, object library, OS built-in facility, other existing components, or as a composite component implemented as another architecture. The interface of a component is typically specified by the following properties:

- *Type*: type of the component.
- *Port*: one or more ports supported by the component type
- *Constraints*: constraints imposed on the ports of the component

Similar to the use of types for capturing properties of data in programming languages, one can define component types as an abstraction that encapsulates certain recurring properties of components. Specifically, component types are intended to capture the architectural properties. For example, the type *filter* specifies that a component must be used in a pipe-and-filter system, whereas a component of type *module* should be used in a main program/subroutine system. A component can be used as a black box with a set of ports, each of which defines a

behaviour that a component may exhibit in constructing a system. Although the term "component-based development" does not refer to the development of just one type of system, or to the use of just one type of component, most conceptions of component-based development share a few fundamental concepts. Foremost of these is that components are software implementations that have interfaces and that are units of independent substitution. But beyond this, there are numerous variations, both large and small.

For the context of this report, the following categorization of components was adopted. *Components Off the Shelf* (COTS) and *Free Open Source Software* (FOSS) components.

1.2 Component of the Shelf (COTS)

Software engineering, as a general domain, still lacks the precision and clarity that can be found in other engineering and science domains. Similar to the ambiguous definitions of what generally constitutes a component, one faces a similar situation in defining COTS. There exists a large body of research which deals with components and more specifically COTS. Common to most of them is that most of these articles adopt their own, new definition of what a Component Off The Shelf is. The following is a brief overview of some of the COTS definitions found in the literature [MOR02].

Oberndorf

In [OBE97], the term COTS product is defined on the basis of the 'Federal Acquisition regulations'. It is defined as something that one can buy, ready-made, from some manufacturer's virtual store shelf (e.g., through a catalogue or from a price list). It carries with it a sense of getting, at a reasonable cost, something that already does the job. The main characteristics of COTS are: (1) it exists a priori, (2) it is available to the general public or (3) it can be bought (or leased or licensed). The meaning of the term "commercial" is a product customarily used for general purposes and has been sold, leased, or licensed (or offered for sale, lease or license) to the general public. As for the term "off-the-shelf", it can mean that the item is not to be developed by the user, but already exists.

Vidger

The work of Vidger and colleagues, presented in [VIG96], provides a different definition of COTS products. They are pre-existing software products; sold in many copies with minimal changes; whose customers have no control over specification, schedule and evolution; access to source code as well as internal documentation is usually unavailable; complete and correct behavioral specifications are not available

SEI

According to the perspective of the SEI, presented in a recent work [BRO00], a COTS product is: sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; supported and evolved by the vendor, who retains the intellectual property rights; available in multiple, identical copies; and used without source code modification.

Basili and Boehm

More recently, Basili and Boehm [BAS01] proposed another definition of COTS. According to their definition, COTS software has the following characteristics: (1) the buyer has no access to the source code, (2) the vendor controls its development, and (3) it has a nontrivial installed base (that is, more than one customer; more than a few copies). This definition does not include some kind of products like special purpose software, special version of commercial software, and open source software. The category of products addressed by such definition presents some specific non technical problems, related to the quick turnaround (every 8-9 month) [Bas01] of product releases. In addition, marketplace consideration adds further variability: in the COTS products market, there are no widely agreed upon standards mainly due to marketing strategies aimed at obtaining vendor lock-in. Variability and marketing strategies suggest that there will never be a single unified marketplace of standardized COTS products [WAL02].

Rather than introducing a completely new definition of a COTS component, the following definition provided by the CeBASE group in Maryland is adopted and extended for the purpose of this report.

Definition COTS [www.CeBASE.org]:

- developed by a third party (who controls its ongoing support and evolution),
- bought, licensed, or acquired,
- for the purposes of integration into a larger system as an integral part, i.e. that will be delivered as part of the system to the customer of that system (i.e. not a tool),
- which might or might not allow modification at the source code level,
- but may include mechanisms for customization,
- and is bought and used by a significant number of systems developers.

We refine the above definition by including the following additional property:

- COTS based systems do not provide source code that can be analyzed or modified.

Typical COTS-based systems (CBS) include commercial-off-the-shelf (COTS) products besides newly written (in-house) software and/or FOSS components. The most important features of COTS products are their suitability for integration into different systems and commercial availability. These aspects allow COTS products to provide high quality pre-packaged functionalities. The relationship between components and COTS is strong, but COTS and components should be considered as two different concepts. In summary, one can say that COTS products and components are two sets with a non-empty intersection but both lack a clear definition.

COTS based development can be summarized as fundamentally the problem of integrating black-box components rather than building components. This integration process is not easy. It is error prone, requires significant amount of coding, and is difficult to test and debug. In addition, many COTS components have a high level of volatility. Commercial components are often subject to frequent upgrades. These upgrades may not have the added functionality/bug fixes desired by the integrator. Critical functionalities which existed in a previous version may have been removed in a subsequent upgrade. In some cases the integrator may wish to substitute similar components from different vendors in new releases of the system. The following list is a summary of challenges typically associated with COTS [HAD02].

- Plug-and-play of components. The architecture of the system must allow the substitution of components. Component substitution can involve substituting one version of a component for a different version, or substituting a component with similar functionality from a different vendor.
- Decoupling between components. There must be minimal coupling between components. Coupling can involve both functional coupling, such as procedure calls, as well as other dependencies such as resource contention or architectural assumptions. The architecture must allow for the isolation of components.
- Hiding unwanted functionality. In order to differentiate their product from competitors, COTS vendors often overload their systems with a large amount of functionalities. Far from being an advantage in the COTS based system, the system architect may wish to remove this functionality. Since this cannot be done with a COTS component, the architecture must provide designers with a mechanism for masking the unwanted functionality so that it is inaccessible to the end-users and/or the system programmers.
- Debugging and testing. Since COTS components are black-box it is impossible to access their internals for the purposes of testing or debugging. An architecture and design cannot eliminate this problem, but it can include the capability of monitoring and verifying component behavior during runtime, and preventing faults in a component from propagating through the system.
- Evaluation requirements: COTS products must be evaluated to decide whether or not they are suitable for the substitution in a given environment.
- Integration/substitution: In-house software and COTS products have their own assumptions regarding interfaces, packaging, functionality, etc.; if these assumptions are different, integration work will be necessary to make COTS products work in the system.
- Security: COTS can provide a source of a security hazard, permitting a dangerous behavior due to a Trojan horse or just an accidental failure.
- COTS component source code is typically unavailable; thus, the component must be analyzed and tested as a "black box."

- Updates and evolution of a COTS component are provided by the vendor. New functionality of an updated component could be detrimental to specific applications that use it. In fact, functionality in the original component could also be problematic.
- The vendor often fails to provide a correct or complete description of the COTS component's behavior. This can result in the buyer of the component having to guess how the component is meant to be used or how it is supposed to behave. Worse yet, the buyer could end up using the component in a manner the vendor did not intend. Unanticipated uses could compromise the reliability of both the COTS component and the application into which it is integrated.
- Maintenance can become an issue because the vendor may not correct defects or add enhancements as the buyer needs them. Developers in the organization that purchased the component may be forced to make modifications themselves, which can be quite difficult if the component's source code is unavailable or if the component's specification is poor. Clearly, the creation or maintenance of systems that use COTS components is by no means a trivial endeavor. On the contrary, integrating COTS components into an application is prone to error, can require a significant amount of coding, and can be problematic to test properly

1.3 Free and Open Source Software (FOSS)

Software in general can be classified as either proprietary or free and open source software. The basis for this distinction is based on the rights to which the user of the software is entitled. Proprietary software typically denies rights that the user has with free and open source software, the most prominent being the right to look into (and thus learn about) the internal mechanisms of the software and others being the right to change the way in which the software operates, the right to let others see or use the software, etc. Companies producing proprietary software sell users very limited rights in an effort to maintain control over the software. Some examples of these are Microsoft Windows 2000/XP, Macromedia Dreamweaver, Adobe Photoshop, etc. A somewhat polemic statement often heard in this context is: "Free and open source software is not *free* as no cost, but *free* as in *freedom*." The rights granted for free and open source software are typically laid out in an accompanying license, but that license will not demand any fees or payments for the basic rights discussed above. A great number of licenses have appeared all claiming to be open source licenses. This has prompted the open source community to identify guidelines for judging whether a license should be considered free and open source software [PER98].

The use of the rights to the source code can be used to classify components into a category of Free and Open Source Software Component category. This category is also referred to as **F/OSS** or **FOSS**. A basic characteristic of this type of components is that the software or component is liberally licensed to grant the right of users to study, change, and improve its design through the availability of its source code. F/OSS is generally synonymous with free software and open source software, and describes the same licenses, culture, and development models.

The fundamental differences between FOSS and proprietary software (COTS) give rise to many phenomena about FOSS that have recently been studied and reported. With its source code available for everyone interested, a culture of peer review and criticism (not unlike that known from academia) has evolved around FOSS. Participation is encouraged based on the common goal of improving the software. Contributions receive high esteem and contributors take pride in their contribution. A more appropriate categorization for FOSS would be the level of expertise and the degree of participation by a user/developer. Given enough time and skilled employees, an organization can verify that the open source software it uses does exactly what the organization wants it to do. While most organizations will not want to invest this kind of effort, if the software is found defective in a specific aspect and that aspect is of sufficient importance to the organization, the organization can for example, order employees to correct the defect. Some examples of FOSS are Linux (the operating system named after Linus Torvalds, who started developing the system in 1991 as a student of Helsinki University), Apache (web server), Mozilla (Internet browser), MySQL (database suite), Open Office (office software) and many more.

During the past two decades, the software market has been dominated by Commercial-Off-The-Shelf (COTS) products that offer a myriad of functionalities at reasonable prices (e.g. Microsoft Windows and Oracle database management systems). However, the intrinsic limitations of COTS software have emerged over time (i.e. closed source code, expensive upgrades, lock-in effect, security weaknesses, etc.). This led to the development of a parallel "economy" based on free and open source software. FOSS development has grown. Thousands of FOSS projects are now carried out via Internet collaboration; hundreds of high quality applications are available for use or modification at no (or small) cost; and many FOSS products are now widely available that are considered to be as mature and secure as their COTS equivalents. It has been suggested that the high level of quality found in some free

software projects is related to the open development model which promotes peer review. While the quality of some free software projects is comparable to if not better than that of closed source software, not all free software projects are successful and of high quality. Even mature and successful projects face quality problems, some of which are related to the unique characteristics of free software and open source as a distributed development model based primarily on volunteers.

In the context of this survey the following informal definition of a FOSS component is adopted.

Definition FOSS
A Free and Open Source Software (FOSS) component is characterized by:

- developed by a community of developers who control its ongoing support and evolution,
- free of cost,
- licensed with right of users to study, change, and improve its design through the availability of its source code licensed, or acquired,
- the ability to modify the underlying source code,
- it is used by a significant number of systems developers.

1.4 Component Classifications Applied in Survey

As mentioned earlier, component substitution is a multi-dimensional problem space. Component substitution can be informally defined, as the process of replacing an existing component with either its equivalent or a new component that meets the new functional or non-functional requirements.

1.4.1 COTS versus FOSS Substitution

As defined in the previous section, two major categories of components are distinguished to simplify the structure of this report: COTS and FOSS components. COTS components being licensed third party components without any source code available, with only the binary or executable available. For the FOSS based components on the other hand, source code and other code related information, like test cases, etc. are available. FOSS components are open source and are both freely available and modifiable. Based on these two categories, different component substitution combinations can be identified. Table 1 illustrates these possible substitution combinations. The focus in this report is on the techniques and information needs, as well as the information provided by these techniques that are required to perform a component substitution for one of the highlighted substitutions in Table 1.

Table 1 Software component substitution types

FOSS	**Substitute with**	**FOSS components**
FOSS	Substitute with	COTS components
COTS	**Substitute with**	**FOSS components**
COTS	Substitute with	COTS components

1.4.2 Categorization of Abstraction Levels for the Component Substitution

Components from an architecture view can be seen as a functional decomposition of the system, i.e. the functions of the systems are assigned to different components. Initially, such an assignment is often performed implicitly during the specification and design phase. However, during the implementation and maintenance phase, this initial assignment might change. Also, the meaning of a relationship between components may vary and is often defined implicitly. The relationships may imply functional dependencies, i.e. component uses and depends on the provided functions of the other component. The relationships may also indicate synchronization, data flow or some other kind of dependency. Dependencies to other components are very valuable as part of performing component substitution. Hence, the required interfaces for the components are among the first to be considered. In general, the more information and explicitly defined semantics of the relations are available, the more accurate results can be obtained by techniques and tools performing modifiability analysis.
A typical modeling aspect of software components is therefore the granularity at which components are defined or expressed. While classic software development was based on homegrown applications, in the 1990s Enterprise

Resource Planning (ERP) systems emerged, in which functionally complete subsystems are considered as basic components and an information system is developed by assembling and customizing these components. ERP suites provide a single, homogeneous solution for a significant number of back-office functions in an organization, such as integrated finance, human resources, and manufacturing/supply-chain processes, by defining common semantics, common models for the organization, and a single architecture.

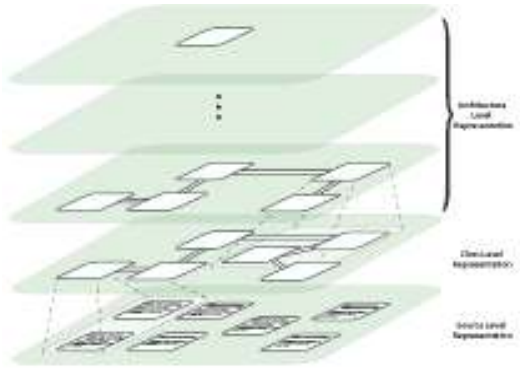


Figure 1. Component abstraction levels

More recently, there has been a trend is towards finer-grained components and application frameworks. Examples are given by the various e-commerce component suites and frameworks offered by different vendors, in which components are used as building blocks for assembling new e-commerce applications and portals. The major obstacle to the establishment of component based software is the need for a common framework. Recently, new emerging technologies, such as the Internet based ones and the eXtensible Mark-up Language (XML), and distributed object computing (DOC), provide the technological backbone for the effective development of distributed components. They are based on the merge of distribution and middleware technologies and object orientation. The computation is performed through messages that software objects – developed in different programming languages and deployed on heterogeneous hardware and software platforms, e.g. OMG CORBA Component Model (CCM), the Enterprise JavaBeans (EJB) architecture and the Microsoft Component Object Model (COM+). In its most basic and low level representation, a component can correspond to a set of statements or a function that was programmed to allow for their reuse or interaction with other programs. The granularity levels shown in table 2 are applied to classify the component substitution techniques.

Table 2 Component abstraction levels

Abstraction level	Scope and description
• Statement level	Basic techniques from low level analysis at the source code/statement level that build to some extend the foundation for the higher level analysis techniques.
• Class/function level	At this abstraction level the focus is on interaction among methods, classes.
• Feature/package	The analysis and techniques applied in this section identify interactions and dependencies among large artifacts, allowing for interpretations that go beyond just traditional structural information or dependencies
• Subsystem/System level	The motivation at this level is to provide different viewpoints and interpretations of systems or subsystems, their interactions and dependencies. They also often allow to infer domain knowledge or user expertise
• Documentation/Traceability	This category focuses on the analysis on documents supplementing components (e.g. design, requirement documentation) and the traceability among these and the source code.

1.4.3 Component substitution process

The IEEE Standard Glossary of Software Engineering Terminology [IEE90] defines software maintenance as "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment." This definition further describes three types of maintenance: 1) *Corrective Maintenance* that modifies the software to fix defects; 2) *Adaptive Maintenance* that consists of modifications made to keep the system usable after changed or changing environment; and 3) *Perfective Maintenance* that improves performance or maintainability.

Component substitution can be seen as a specific instance of a maintenance activity. From that perspective a component substitution process will have similarity with the more general maintenance models and process models. The IEEE Standard for Software Maintenance [IEE98] clearly defines 7 key steps that are involved in maintaining software (these steps are shown in Figure 2). The first step is to identify, classify and prioritize the problem. This assumes an understanding of the system to be maintained. This step is followed by a feasibility analysis that will, among other things, determine the impact of the change. Steps three to seven of the maintenance process involve implementing the change and ensuring the quality of the changed system.

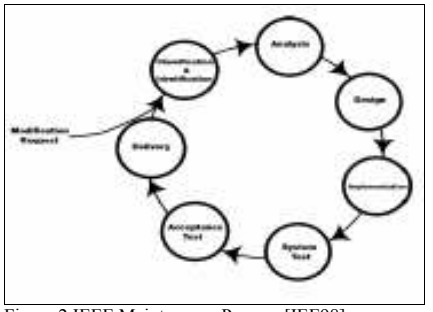


Figure 2 IEEE Maintenance Process [IEE98]

In this report the focus is on component substitution and the identification and comprehension of the system, the component to be substituted and the information and techniques that have to be applied to perform the substitution process. In larger systems, the identification and comprehension of components as well as their interactions with other components and the system in which they are integrated/substituted becomes a major challenge. The substitution process is affected by the component type (COTS versus FOSS), the abstraction level of the component and the techniques applicable to support the comprehension and substitution of these components. Furthermore, it also involves conflict identification, component adaptation, and implementation of connections between components. Usually, an existing component does not exactly match the requirements of a new system. Component adaptation is required in this case in order to resolve the conflicts (or mismatches) between an existing component and a requirement. For instance, a typical mismatch is that the order of arguments to an existing procedure may be different from that of a query specification, even though the procedure can satisfy the query functionally.

Any component substitution process will therefore have to consider different artifacts (software are non-software), e.g. domain model, the program model and the current understanding a maintainer has gained of the system. However, these models will be affected by the abstraction level and component type (COTS vs. FOSS) the substitution is performed on.

There is a clear need to have a component substitution process model in place to avoid some of the problems that typically are caused by an ad *hoc* approach to component substitution and integration.

1. Conflicts are not easy to identify, since they are usually hidden in the implementation.
2. Component substitution may be quite difficult, particularly for conflicts that are introduced at the design level. For a given component, there typically exist assumptions about the structure of the system in which it will function. These assumptions, called architectural assumptions, constrain the way the component can be

used. For example, in a client/server system, the client expects to interact with the server in a specific way that is very different from what a filter expects in a pipeline system. Thus, a filter in a pipeline system may not be used as a client in a client/server system, even though it may have the required functionality. This conflict is termed an architectural mismatch, recently identified as a serious obstacle in implementing reuse.

3. Connections between software components are usually treated implicitly, rather than encapsulated and described as explicit entities. This approach increases the complexity of a connection implementation, since the implementation is distributed throughout the implementation of components.
4. The issue becomes even more complex when components have compile time or run time parameters that essentially generate a customized component. Here one has to keep track of which parameter settings relate to which version, and to which set of selected features.
5. Code that implements a particular feature or several closely related features needs to be spread across multiple products, subsystems, modules or classes and is intertwined or tangled with code implementing other (groups of) features. In many (ideal) cases, a particular feature will be implemented as code that is mostly localized to a single module; but in many other cases features cut across multiple components. Such "crosscutting" concerns make it very difficult to associate separate concerns with separate components that hide the details. Examples of crosscutting features include end-to-end performance, transaction or security guarantees, or end to end functional coherence. To produce a member of the product line that meets a particular specification of such a cross cutting feature, requires that several (or even all) subsystems and components make compatible choices, and include specifically related fragments of code.

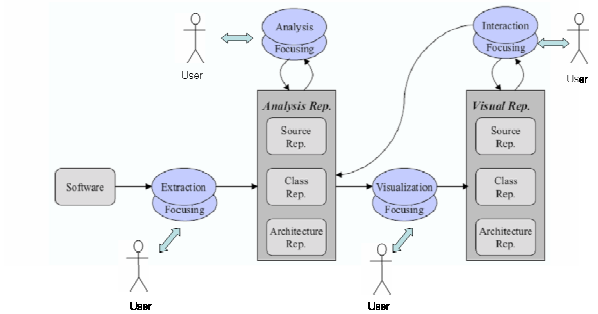


Figure 3. Substitution process

As illustrated in Figure 3, the component substitution process is a multi dimension process requiring the incorporation of multiple information artifacts. The substitution process itself can be performed at various levels of abstractions as discussed in section 1.2.4. common to all these information needs and techniques is that the ultimate goal is to provide maintainers with sufficient information to perform a component substitution without violating the behavior and functionality of the existing system. Critical design decisions must be made in the face of incomplete and often mistaken understanding of the features and behavior of software components and knowledge gaps are inevitable and are a major source of risk during a component substitution.

- Whatever knowledge is obtained about one commercial software component does not translate easily to components from different vendors, and all component knowledge tends to degrade quickly as components evolve through new releases.
- Competitive pressures in the software marketplace force vendors to innovate and differentiate component features rather than stabilize and standardize them. This results in mismatches that inhibit component integration and inject significant levels of wholly artificial design complexity.
- Use of commercial components imposes a predisposition on consumers to accept new releases despite disruptions introduced by changing component features. These disruptions take on a random quality across all phases of development as the number of components used grows. These challenges all derive from the same root cause: a loss of design control to market forces.

1.5 Outline

This state of the art report on component substitution includes 5 major chapters. The general outline is shown in Figure 4. Chapter 1 provides a general introduction to the relationships between component substitution, software maintenance and program comprehension. It also provides definitions of the basic terminology and provides a general outline of the classifications used to categorize components, the substitution abstraction levels and the process involved in performing the substitution process.

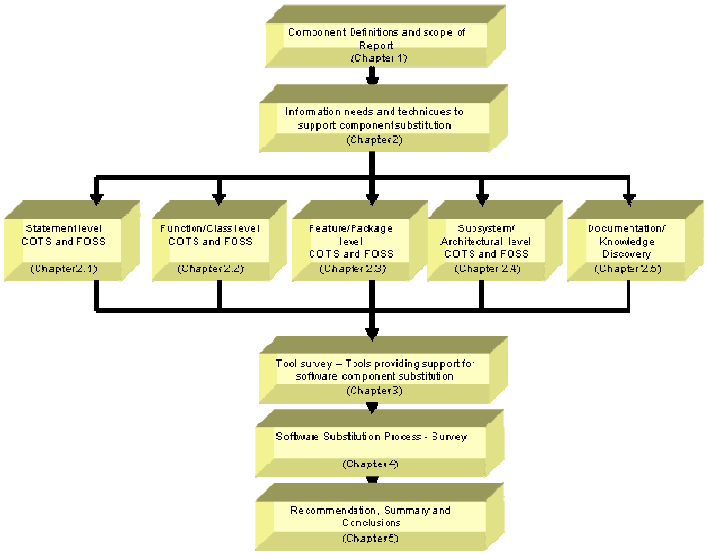


Figure 4. Report Outline

Chapter 2 corresponds to the main part of the report, which presents a state of the art review of techniques, their information needs and the information provided by these techniques to support component substitutions. The review is divided into the different abstraction levels: Statement, function or class level, feature/package level and subsystem/system level. Furthermore, each of these categories is divided into COTS and FOSS components, due the fact that the techniques available for these different types of components differ significantly. The review also includes a separate category related to documentation, knowledge discovery and traceability as an information resource, other than source code or the component itself. Chapter 3 provides an initial tool survey of tools supporting techniques relevant to component substitution at the various abstraction levels. It should be noted that the tool survey in this chapter is only an initial survey and excludes tools typically found and covered in other surveys in the reverse engineering, program comprehension and software visualization domain. Chapter 4 discusses some of the existing process models supporting software maintenance or component maintenance. A brief discussion on the challenges and open issues for developing a component substation process are discussed. Chapter 5 will conclude with an overview of the surveyed techniques, some recommendations and conclusions.

2 Component Identification and Comprehension

Software comprehension plays a dominant role in most software maintenance activities. Typically between 60-80% of the total cost associated with software maintenance is spent on activities directly related to comprehension. As outlined in section 1, software component substitution can be seen as a special instance of a software maintenance task. Similar to most software maintenance tasks, component substitution is not a well-defined process, due to its dependency on various factors, like the granularity of the substitution, the level of abstraction at which the substitution is performed, the type of system, a user's expertise, etc. . This multi-dimensionality of the problem requires information form various resources and techniques that have to be applied at different levels of abstraction to support a component substitution. As part of a component substitution there is a clear need to identify and comprehend the system and the components as well as their interactions with other system artifacts and components.

As within traditional software maintenance activities, reverse engineering can be used to obtain higher level representations of programs to gain a general understanding of a system. Reverse engineers typically start with a low level representation of a system (such as binaries, plain source code, or execution traces), and try to distill more abstract representations from these (such as, for the examples just given, source code, architectural views, or use cases, respectively). Reverse engineering and its underlying techniques not only allow for the identification of components, but also for the identification of the resulting interconnections among components and subsystems. Gaining this understanding builds an important step towards establishing a successful component substitution process. Techniques applied during reverse engineering and program comprehension typically fall in one of these four categories.

- *Textual, lexical and syntactic analysis* – Common to these approaches are their focus on the source code and its representations. Most of these techniques in this category are based on static analysis, where the source code is parsed and stored into an inter-media format/representation (e.g. Abstract Syntax Tree), and then further represented at various levels of abstractions or views. The stored information typically can be retrieved using provided query and retrieval techniques associated with the intermediate format. Further filtering is performed with graph based analysis approaches, including control, data flow of a program and program/system dependency graphs.
- *Visualization techniques* – Software visualization aims in providing a high-level abstraction/view of the system under investigation, by hiding non-relevant detailed information and focusing on high-level abstract views of the relevant information. Visualization techniques are typically combined with different lexical, syntactical analysis techniques to provide different views and interpretations of the underlying information.
- *Execution and testing* – Techniques in this category are typically dynamic in their nature and based on profiling, testing and observing program behavior, including actual execution and inspection walkthroughs [WEI81, PAG91].
- *Domain knowledge based analysis* – these approaches focus on recovering the domain *semantics* of programs by combine domain knowledge representation and source code analysis. DESIRE [BIG94] and LaSSIE [DBS91] are typical examples in this category. Other techniques try to establish traceability links between source code and problem/application domains.

Although research papers presenting reverse engineering and program comprehension techniques typically describe the steps needed for the successful application of one specific technique, a number of questions remain open. Some of these open questions are: What problems require which technique and information needs? What are typical views that are needed to perform the comprehension task at a specific level of abstraction? Which techniques are suitable for reconstructing these particular views? How can the information extracted be presented so that they actually help to deal with the problem at hand.

In this chapter we address these questions, by focusing the comprehension process on a more specific comprehension subset – the support for both the information and techniques needed to provide support for the comprehension process involved during component substitution. The resulting survey will focus on their information needs with respect to what type of input they require, as well as the information these techniques provide.

The following table provides an overview of the abstraction levels used for the remainder of this chapter, to categorize both the information needs and techniques that can support a component substitution comprehension process.

Table 2 Abstraction levels

Abstraction level	Scope and description
• Statement level	Basic techniques and information from low level analysis at the source code/statement level that build to some extend the foundation for the higher level analysis techniques.
• Class/function level	At this abstraction level the focus is on interaction among methods, classes.
• Feature/package, component level	The analysis and techniques applied in this section identify interactions and dependencies among large artifacts, allowing for interpretations that go beyond just traditional structural information or dependencies
• Subsystem/System level	The motivation at this level is to provide different viewpoints and interpretations of systems or subsystems, their interactions and dependencies. They also often allow to infer domain knowledge or user expertise

2.1 Statement level

Software component substitution relies on the ability of tools to provide and extract higher-level representations (i.e., software architecture) from existing software systems. The motivation is to support engineers in assessing, maintaining, and evolving large-scale software systems and their components. In this section of the survey the focus in on techniques at the statement level and their information needs. These techniques build often the basis for other techniques at higher levels of abstraction (e.g. class, feature, architectural level). The section is divided as all of the remaining sections in this chapter in the two main categories, COTS and FOSS based techniques.

2.1.1 COTS

Based on the previously introduced definition of a COTS based system, COTS are typically provided as a blackbox component, with no source code available for inspection or analysis. The unavailability of source code limits both the information and the number of techniques that can be applied to support the comprehension of components at this level. Among the few techniques available one can consider the de-compilation of components. Many tasks in the area of code analysis, manipulation and maintenance require a control flow graph (CFG) at the statement level, like program slicing. The following table provides an overview of the COTS based statement level techniques covered in this section.

Table 3 Overview COTS analysis at the statement level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Decompilation	Bytecode	Assembly code, Java code (decompiled)	Flow analysis Dependency analysis	Y	N
Instrumentation (virtual machine), bytecode	Bytecode	Execution traces	Dynamic flow and Dependency analysis	N	Y
Flow analysis (mainly control flow), Slicing	Decompiled assembly or Java Code	Control and data flow, limited slice	Flow analysis Dependency analysis	Y	Y

2.1.1.1 Decompilation techniques:

Decompilation is similar to disassembly, by taking source code apart. While disassembly translates executable code into assembly language, which is a close approximation to machine language, decompilation translates the same executable code to a high-level language that will require little knowledge of the computer on which it will be run. Compiler-writing techniques are well known in the computer community. Decompiler-writing techniques are not as well yet known. The structure of decompilers is based on the structure of compilers; similar principles and techniques are used to perform the analysis of programs. The first decompilers appeared in the early 1960s, a decade after their compiler counterparts. A decompiler is a program that reads a program written in a machine language, the source language, and translates it into an equivalent program in a high-level language, the target language (see Figure 5). There is a general understanding that a meaningful disassemble of a machine code program (recover the assembly sources; this in itself is not a trivial problem) is not feasible. However, in practice, there have been approaches to deal with disassembly and decompilation of machine code. The main problems with decompilation are the separation of data and code (i.e. obtaining a complete disassembly of the program), the reconstruction of control structures, and the recovery of high-level data types. In order to achieve a greater percentage of the disassembly automatically, decompilers can make use of knowledge about certain compilers and libraries used in the compilation of the file to be decompiled. The more successful approaches will make use of extra information (e.g. knowledge of the compiler used) or require human input for the difficult parts of the disassembly process.

Figure 5 Decompiler overview

From a component substitution view, a decompiler is an aid in the recovery of lost or not available source code (in the case of COTS), or the migration of applications to a new hardware platform. Typical application of decompilation are in the translation of code written in an obsolete language into a newer language, the structuring of old code written in an unstructured way (e.g. ‘spaghetti’ code) into a structured code, and as a debugger tool that helps in finding and correcting bugs in an existing binary program. Furthermore, from a security point of view, a binary program can be checked for the existence of malicious code (e.g. viruses) before it is run for the first time on a computer, in safety-critical systems where the compiler is not trusted, the binary program is validated to do exactly what the original high-level language program intended to do, and thus, the output of the compiler can be verified in this way.

However, a decompiler writer has to face several theoretical and practical problems when writing a decompiler. The decompiler typically has to be developed for the specific environment (operating system, programming language, and processor) in which it will be used. Other limitations include that any meaningful names that programmers give to variables and functions (to make them more easily identifiable) are not usually stored in an executable file, so they are not usually recovered in decompiling. However, the main problem derives from the representation of data and instructions in the Von Neumann architecture: they are indistinguishable. Thus, data can be located in between instructions, such as many implementations of indexed jump (case) tables. This representation and self-modifying code practices makes it hard to decompile a binary program. Also, most operating systems do not provide a mechanism to share libraries. Consequently, binary programs are self-contained and library routines are bound into each binary image. Library routines are either written in the language the compiler was written in, or in assembler. This means that a binary program contains not only the routines written by the programmer, but a great number of other routines linked in by the linker. As an example of the amount of extra subroutines available in a binary program, a ‘hello world’ program compiled in C generates 23 different procedures. The same program compiled in Pascal generates more than 40 procedures.

Some of these problems can be solved by the use of heuristic methods, others cannot be determined completely. Due to these limitations, a decompiler performs automatic program translation of some source programs, and semi-automatic program translation of intermediate formats of other source programs. This differs from a compiler, which performs an automatic program translation of all source programs. Results can be improved if the decompiler has the entry points of the libraries and DLLs available, as well as an editable symbol table used by the original compiler to help the decompiler to document its output.

One exception in which the decompilation of executable bytecode is somewhat easier is Java. The Java virtual machines execute Java bytecode instructions. Since this bytecode is a higher level representation than traditional object code, it is possible to decompile it back to Java source. Many such decompilers have been developed and the conventional wisdom is that decompiling Java bytecode is relatively simple. This may be true when decompiling bytecode produced directly from a specific compiler, most often Sun's javac compiler. In this case it is really a matter of inverting a known compilation strategy. However, there are many problems, traps and pitfalls when decompiling arbitrary verifiable Java bytecode. Such bytecode could be produced by other Java compilers, Java bytecode optimizers or Java bytecode obfuscators. Java bytecode can also be produced by compilers for other languages, including Haskell, Eiffel, ML, Ada and FORTRAN. These compilers often use very different code generation strategies from javac.

2.1.1.2 Bytecode Instrumentation

The heterogeneity and dynamism of today’s software systems make it difficult to assess the functionality, performance, correctness, or security of a system outside the actual time and context in which it executes. As a result, there is an increasing interest in techniques for monitoring and dynamically analyzing the runtime behavior of an application. In general, when collecting dynamic information, one is interested in collecting such information for some specific entities in the code (e.g., method calls and paths) and in a subset of the program (e.g., in a specific module or set of modules). More precisely, an instrumentation task has to specify (1) which instrumentable entities to instrument, (2) the parts of the code in which those entities must be instrumented, (3) the kind of information to collect from the different entity types, and (4) how to process the information collected.

Instrumentation tools are generally not aware of the semantics of information passed via the annotation mechanism. This is especially true for post-compiler, e.g., runtime, instrumentation. The problem is that instrumentation may affect the correctness of annotations, rendering them invalid or misleading, and producing unforeseen side-effects during program execution. It should also be noted that any type of instrumentation might lead to non-deterministic effects, which might lead to a program behavior that would normally not occur in the un-instrumented version of the same program.

Modern programming languages, as well as traditional ones, support program annotation (e.g. Java). Annotations constitute a powerful mechanism that enables passing information between programmers, tools, and the runtime, from the source code level up to the execution time. Program annotations enrich the program semantics and facilitate optimizations. They describe method usage (test markers, web method markers), convey optimization hints (static register allocation schemes, redundant runtime checks mark-up), or aid in code development and maintenance (author name tags, bug tracking, analysis of program behavior). In Java, annotations were only standardized recently in release 1.5. *Java Instrumentation Services* standardize the means by which a Java “agent” can query and modify the JVM in which it is running, and in particular, instrument at runtime classes running in that JVM. The Instrumentation Services were standardized in the same 1.5 release of Java as the annotations. For Java programs, two common approaches for collecting such information are to add to the code ad-hoc instrumentation using a bytecode rewriting library or to leverage capabilities of the runtime systems (e.g., the Java Virtual Machine profiling or debugging interface. Unfortunately, these approaches are usually expensive and result in experimental infrastructure that is not extensible and, thus, hard to reuse and modify. Another approach for alleviating the problem of collecting runtime information is to use an aspect-oriented language. Although aspect-oriented languages provide a convenient mechanism for inserting probes at specific points in a program, they are often inadequate: first, existing aspect-oriented languages are not able to provide certain kinds of information, such as information at the basic-block level; second, aspects can incur a large time overhead.

Studies on .NET have shown that two-thirds (66%, or 143 out of 215) of the nonstandard annotations implemented by this platform target the runtime. There is also a growing body of annotations consumed at runtime by third-party tools such as test drivers. Instrumentations are commonly used to track application behavior: to collect program profiles; to monitor component health and performance; to aid in component testing; and more. As opposed to general-purpose program transformations, instrumentation only aims to gather additional information about the system rather than modify the original program’s structure and behavior. Bytecode instrumentation uses structural and semantic information provided by language and platform specifications both to identify instrumentation points and to avoid affecting the original program structure and behavior. It has to be noted that .NET provides a non-standard profiling APIs for run-time program transformation and instrumentation.

Java Bytecode Instrumentation

The *Java Virtual Machine* (JVM) known as a runtime interpreter is the component of Java that is responsible for its hardware and operating system independence. The JVM is an abstract computing machine. Like a real computing machine, it has an instruction set, known as JVM bytecode [LIN96]. The Java source code is normally compiled in a binary format to a bytecode instruction set (i.e. the class file) as an intermediate format. The Java bytecode is hardware independent and operating system independent. So the high level meaning of Java source code is first transformed by the Java compiler to this intermediate representation before execution.

Java bytecode instrumentation, also called bytecode injection, bytecode insertion or class file transformation, is the process of directly inserting or manipulating Java bytecode. The instrumentation of Java bytecode generally inserts a special, short sequence of bytecode at the designated points in a Java class file. This transformation process must be strict and should adhere to the constraints imposed by the JVM Specification on the Java class file format, so any modification of JVM bytecode should be reflected on all other JVM bytecode within a Java class file.

The bytecode instrumentation can be performed either statically at compile time or dynamically at runtime. The static instrumentation of bytecode can occur during or after compilation of a Java source file. The instrumented bytecode is saved in a class file, like typical Java classes files, and then executed later by the JVM. The dynamic bytecode instrumentation takes place at runtime. A typical way to perform runtime instrumentation is to insert bytecode into a Java class when the bytecode of the class is being loaded into the JVM. The dynamic bytecode instrumentation can also be applied at runtime to redefine a loaded class or create a new class from scratch.

The Java bytecode instrumentation allows to perform dynamic analysis of those instrumented classes, mostly for debugging, testing, profiling, monitoring and comprehension of dynamic behavior.

2.1.1.3 Data and Control Flow analysis [KIS05]

There exists quite a significant body of research on program slicing, a comprehension technique which is best known for its ability to compute source code slices for programs written in a high-level language. However, comparatively little attention has been paid to the slicing of binary executable programs. However, there are some potential applications for slicing of programs without source code to address for example security concerns. The detection of such malicious code fragments, especially in COTS components, is starting to become a major concern of researcher. Cifuentes and Frabuolet [CIF97] presented a technique for the intraprocedural slicing of binary executables, but no usable implementation of this interprocedural solution can be found. Bergeron et al. [BER99] suggested using dependence graph-based interprocedural slicing to analyze binaries. However no discussion of problems that might arise or concrete experimental results are provided. Many tasks in the area of code analysis, manipulation and maintenance require a control flow graph (CFG). It is also necessary for program slicing to have a CFG of the sliced program as every step in the slicing process depends on it. However, the control flow analysis of a binary executable has a number of problems associated with it. In a binary executable the program is stored as a sequence of bytes. To be able to analyze the control flow of the program, the program itself has to be recovered from its binary form. This requires that the boundaries of the low-level instructions from which the program is constructed be detected. On architectures with *variable length instructions* the boundaries may not be detected unambiguously. A typical example for this is the Intel platform. Figure 6 shows an example byte sequence interpreted in two ways. This highlights the problem that it has to be detected exactly where the decoding of offset of one byte can and will yield completely false results. On other architectures where *multiple instruction sets* are supported at the same time the problem is to determine which instruction set is used at a given point in the code. If the binary representation *mixes code and data*, as is typical for most widespread architectures, their separation has to be carried out as well.

Static slicing is in general regarded as a useful analysis method for maintenance and program understanding because the irrelevant parts of the program can be 'sliced away'. However, static slices of even small programs are in many cases are still too large. When applying interprocedural static slicing on real life binary executables, an average slice reduction can be reached of 56–68%. Static slices tend to be too large and alternatives are needed to compute slices that are smaller and therefore more useful in practice. Union slicing attempts to address these issues by obtaining a more precise slice by computing the union of dynamic slices. This union of dynamic slices is a good approximation of the real dependences if enough test cases are used. The drawback of this approach is that it is very expensive as it is based on the computation of dynamic slices. The computation of the complete trace makes the execution of programs slower by orders of magnitude and, moreover, in the case of real life programs the trace size can be huge. It was also suggested using dynamic points-to data to reduce the size of static slices of C programs. Dynamic points-to data analysis can make the static call graph more precise by using dynamic information and applied the so-reduced call graph during the slice computation. Results have shown that, in case of C programs

which intensively use indirect function calls, the size of slices can be markedly reduced. Naturally, a slice computed this way is regarded as unsafe, i.e. there may be dependences in the sliced program which are not present in the slice. Nevertheless, a big advantage of this approach is that the reduced slices can be computed with little effort. Moreover, there are applications of slicing where the 'safe' property of such slices is not critical. For example in debugging one could start with a reduced slice and only if the problem cannot be identified with the help of it is the usually substantially larger static slice to be used.

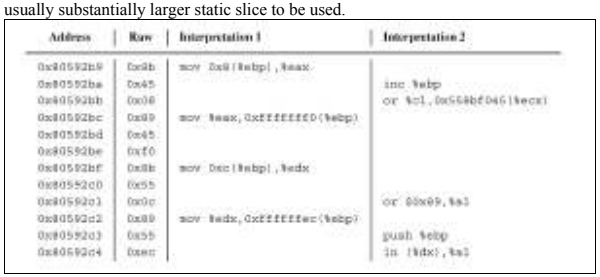


Figure 6 Byte representation [KIS05]

After identifying the instructions of a program, a control flow graph can be created. First, the *basic blocks* are determined, which will constitute the nodes of the CFG, then the blocks are further grouped to represent *functions*. Additionally, for each function a special node, called the *exit node,* is created to represent the single exit point of the corresponding function. The nodes of the CFG are connected with *control flow*, *call* and *return edges* to represent the appropriate possible control transfers during the execution of the program. This requires the behavior analysis of machine instructions. In high-level languages, only indirect function calls fall in this category, but on binary level, intraprocedural control transfer may be represented in this way as well. Another type of problems is when control is transferred between functions in a way different from function calls. *Overlapping* (or multiple entry) and *cross-jumping* functions, which usually do not occur in high-level languages and result from aggressive interprocedural compiler optimizations, are typical examples of this problem. In the case of these constructs, the exit node of the control transferring function is not reached; a control flow edge has to be inserted between the exit nodes of the functions to make up for it.

Overall, multiple challenges remain with data and control flow analysis at the binary level. Firstly, the challenges of identifying the proper offset and the adjustment to the particular compiler used to create the executable as well as the hardware the software was compiled for. Furthermore, even after extracting a control and data flow graph, one has to group these nodes to meaningful blocks and interpretation. In particular, for larger binary files, both the complexity of the graph and the analysis and interpretation is going to be an ongoing challenge.

2.1.1.4 Statement level discussion – COTS

The support for COTS component substitution at the statement level is inherently difficult, due to the limitations in interpreting the binary files at this abstraction level. However, some of the available techniques at this level provide essential information used by other more high-level analysis techniques. The following is a brief discussion on the applicability of techniques, information needs and information provided by these techniques to support component substitution.

- *Information needs*: The information needs at this level are typically limited to binary/executable files.
- *Techniques*: Decompilation is only a valid option in certain application and language domains. It works reasonable well for Java with several decompilers being available. In other cases, decompilers have to be customized or specifically written to support the given environment and programming language. They also face challenges with respect to scalability and the precision and interpretability of the information derived. Control flow analysis of a binary executable has a number of problems associated with it as one shall see

below. In a binary executable, the program is stored as a sequence of bytes. To be able to analyze the control flow of the program, the program itself has to be recovered from its binary form. This requires that the boundaries of the low-level instructions from which the program is constructed be detected. After identifying the instructions of the program, a graph can be build with *basic blocks.* These blocks are used to determine the nodes of the CFG, and then the blocks are further grouped to represent *functions*. The nodes of the CFG are connected with *control flow*, *call* and *return edges* to represent the appropriate possible control transfers during the execution of the program. This requires the behavior analysis of machine instructions. Even the high number of the types of instructions may be hard to cope with, but the hardest problem is raised by control transfer instructions, where the *target cannot be determined unambiguously*.

From a more pragmatic point of view, the applicability of the decompiler and slicing techniques is limited due to their scalability and the need for manual interpretation of the results. Profiling and monitoring are viable options to gain a general understanding of the COTS execution behavior. But the typical loss of meaningful names originally used in the source code and the scalability issue (typically large amount of information that is recorded at this level, limit the applicability of this approach at statement level.

- *Information provided*: The information provided by these techniques ranges from control/data flow graphs, to decompiled binaries, to execution traces, with execution traces being the most useful information provided. These traces are often an information source for other, higher level analysis and interpretation techniques. Another advantage of instrumenting binaries is (1) the source code is not required and (2) and even if the source code is available, it would not have to be modified to support the tracing of program executions.
- *Applicability for component substitution*: Decompilation, control/data flow analysis techniques are limited in their applicability for component substitution. Only in cases there the scope of the analysis is rather limited, these techniques might be applicable. In contrast, binary instrumentation has its application in component substitution, due to the following two factors. Most of the instrumentation approaches are automatic and therefore do not requiring any user interaction. Secondly, the resulting traces provide an important information source for other higher-level based COTS (and FOSS) analysis techniques that can support component substitution.

Table 4 Overview of COTS analysis techniques at the execution level

Technique	Applicability for COTS substitution
Decompilation	Limited. Scalability problems, applicability of resulting information is limited due to difficulties in interpreting the information
Flow Analysis (mainly control flow), Slicing	Specific COTS that are rather small in size. Scalability problems. Interpretation of the results difficult
Instrumentation (Virtual machine), bytecode	Useful not only for dynamic flow and dependency analysis, but also as input to other analysis techniques (COTS and FOSS). Scalability issues. Program/COTS behavior changes

2.1.2 FOSS

During the comprehension of FOSS systems, maintainers often spend considerable time and effort in exploring source code, including browsing the code and searching the parts of interest. Studies have shown that maintainers of a large software system spent up to 60% of their maintenance time on performing simple searches across the entire software system. From a component substitution point of view, these techniques are most suitable to analyze component substitutions that focus on areas that are very localized and typically rather small in their scope (several statements, maybe functions). The techniques surveyed at the statement level and their information needs as well as their information provided can be considered the foundation for many software comprehension tools and techniques at level other than just the statement level. These techniques provide in general very detailed and fine grained information of the source code, supporting the analysis of statement dependencies and extraction of facts and program models for further processing, like searching, querying.

Parsing is applied to delivers static source code models that contain source code specific entities such as files, packages, classes, methods, and attributes and the dependencies between them. Dependencies include, class inherits

and aggregates, method calls and overrides, and variable accesses. Profiling on the other hand can be applied to provide run-time data (i.e. method call sequences) for an executed scenario and complements static source code models.

Table 5 Overview FOSS analysis at the Execution level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Parsing, fact extraction, lexical analysis, queries, token analysis, strings and token matching	Source code, language parser	Facts, AST tokens, strings	Lexical analysis, queries, fact extraction flow analysis, token matching D query support (semantic, lexical) basis for most comprehension and reverse engineering tools	Y	N
Dependency analysis, semantic analysis, slicing, impact analysis	AST, tokens,	Control and data flow, limited slice	Dependency analysis, slicing, flow analysis impact analysis	Y	Y
Source code model	Source code, AST, syntactical/lexical analysis	Persistent storage (relational model, object-oriented model or ontology)	Basis for most reverse engineering tools	Y	N/Y
Tracing, profiling, monitoring	Bytecode, AST, source code	Execution traces	Dynamic flow and dependency analysis, profiling	Y	Y
Statement metrics	Source code, dependency analysis	Coverage, identifier complexity	Grouping, comprehensibility, profiling, maintainability	Y	N/Y

2.1.2.1 Parsing and Fact extraction

Implementation specific data (i.e. facts) is obtained by applying static and dynamic analysis techniques. These techniques include, but are not limited to parsing, profiling and fact extraction. This implementation and typically programming language specific data is relevant to support component substitution techniques at both statement level, as well as other at higher abstraction levels.

Fact Extraction

Fact extraction from source code (i.e., finding pieces of information about the system) is a fundamental and often the first step involved in reverse engineering and program comprehension activities. Fact extraction provides high-level reverse engineering analyses or architecture recovery activities with information available at the source code level which is then stored in a fact base. Such a fact base forms the foundation for further analysis tasks that are conducted next, either manually or (semi)-automatically using tools. A common technique for extracting facts from source code is parsing. It has to be noted that for parsing of source code, it is necessary that there exists a language specific parsers for the given programming language. Therefore, parsing becomes more a challenge for mixed language systems or for framework based software systems. In this context these systems are transcend the pure source code level with their own dialects and constructs. For instance, framework-specific statements may appear in source code comments, and configuration files are used to define certain properties of software systems. Typically, such information is removed by pre-processors or is ignored by parsers.

The result is a reduced fact base lacking often crucial information for further higher level architectural analysis tasks.

A standard representation model used to represent these facts is an abstract syntax tree (AST). An AST is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the

operands of the node operators. Thus, the leaves have nullary operators, i.e., variables or constants. In computing, it is used in a parser as an intermediate between a parse tree and a data structure, the latter which is often used as a compiler or interpreter's internal representation of a computer program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax. A parse tree or concrete syntax tree is a tree that represents the syntactic structure of a string according to some formal grammar. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. The classic example of such an omission is grouping parentheses, since in an AST the grouping of operands is explicit in the tree structure.

Another source that can contain relevant facts about a software system is its existing release histories. Release history data of a software project can be obtained from configuration management systems in the form of modification reports or files generated by versioning systems like CVS, containing change data related to source code modifications. These reports can be parsed for relevant data used to identify logical coupling relationships among the different artifacts.

2.1.2.2 Source code query approaches

In general, syntactic approach uses a formal, hierarchical definition of text structure, invariably some form of grammar. Syntactic systems parse text into a tree of elements, called a *syntax tree*, and then search, edit, or otherwise manipulate this tree. Lexical approaches, on the other hand, are less formal and rarely based on a hierarchical structure. Lexical systems generally use regular expressions or a similar pattern language to describe source code structures. Instead of parsing text into a hierarchical tree, lexical systems treat the text as a sequence of flat segments, such as characters, tokens, or lines. The syntactic approach is generally more expressive, since grammars can capture aspects of hierarchical text structures, particularly arbitrary nesting, that the lexical approach cannot. However, the lexical approach is generally better at handling structures that are only partially described or incomplete.

Browsing and searching texts (source code) by specifying keywords or patterns are facilities provided by almost every text editor and software development environment. In order to improve their flexibility and efficiency, almost all of them are based on a lexical structure and restricted to small predictable chunks of text, especially lines.

2.1.2.3 Lexical Queries on Source Code

Patterns specified by users are typical strings of characters, some of which are intended to match themselves (e.g. keywords), and others to match one or more of a set or range of characters. The latter ranges in expressiveness from the simplest wildcard pattern language with two match facilities (one-character and many-character match) through full regular expressions with alternation, sequence, grouping, and iteration.

Regular expression matching

Regular expression matching is typically performed over files on a line-by-line basis. Although many program comprehension tools have been proposed, the family of *grep based* tools is still considered the most frequently used ones [BAA03]. Common to *grep* tools is that they excel at performing a specific task. Consequently, it is easy to specify a search and the results are returned quickly, frequently with a relevant match. When the search fails, little time or cognitive effort is wasted.

Lexical Source Model Extraction (LSME)

Lexical Source Model Extraction (LSME) technique [MUR96] is a lexical approach that analyzes source code without using programming language parsers. LSME provides a lightweight style of source code analysis tool that permits maintainers 1) to use regular expressions to describe patterns of interest in system artifacts; 2) to specify actions to execute when a pattern is matched to part of an artifact; and 3) optionally, to specify operations for combining matched information to compute structural interactions. [MUR96].

As mentioned previously, maintainers often need to search source code not only by specifying keywords or regular expressions, but also by restricting the search to some structural properties of the result. For example, they might want the result to be an argument of a function, or an if-statement. In order to fulfill such requirements, source code query tools usually perform parsing on the source code and thus allow users to perform such structural queries. Many implementations of regular expression matching do not allow for pattern matching across line boundaries. In another word, since lexical approaches lack abilities to capture the structure of source code, they have also

difficulties to search keywords or patterns that are related to particular structures. For example, for lexical approaches it is almost impossible to distinguish a keyword “student” in a comment or in a variable declaration statement.

2.1.2.4 Syntactical Queries on Source Code

Complex queries on source code cannot be addressed by lexical approaches. Paul and Prakash summarized some of these more complex types of queries frequently used by maintainers in [PAU94]. Queries may be based on global structural information at the source code level, e.g. relations between program entities such as files, functions, variables, types etc. Queries can not only be based on statement-level structural information, e.g. looking patterns (e.g. loop statements) that fit a programming plan or a cliché. They can also be based on flow information derived by static analysis such as data flow and control flow analysis or to find statements that may affect the value of a particular variable [WEI81]. In addition, maintainers also want to perform queries based on high-level design information of the software, e.g. to query whether the software implements a particular design pattern [GAM94] or whether it confirms a reflection model [MUR95].

For these complex queries, both syntactic and semantic source code information has to be modeled and query languages according to these underlying models have to be defined. A number of such source code query systems have been proposed, and many of them share a common structure:

- A repository that stores source code information according the model;
- Tools that populate the repository with structural and/or program flow information, such as language parser, static/dynamic analyzers;
- An interface for the user to submit queries and obtain results;
- A query processor that handles queries by retrieving the repository.

These systems can also be distinguished by their underlying source code Meta model and the query language definition, which are introduced in the following section.

Relational Approaches

A number of approaches (CIA [CHE90], grok [HOL96], RPA (Relation Partition Algebra) [FEI98], sgrep [BUL02], etc) have been proposed using a relational model to represent source code structures.
Among them, CIA (the C Information Abstraction) System [CHE90] is one of the earliest source query system. The CIA system extracts relational information from C programs according to a conceptual model and stores the information in a database. The conceptual model for a C program defines both the objects and relationships at a selected level of abstraction. It also serves as a requirements specification for the information abstractor and determines the extent of knowledge available in the database [CHE90] The model defines five types of objects that are used to describe the C language – *File, Macro, Data type, Global variable and Function*. Each object has a set of attributes. For example, a function has attributes such as 1) file it is defined in, 2) data type the function returns, 3) name of the function, 4) whether it is a static function, 5) its start line number and 6) its end line number. The conceptual model does not explicitly define the order of the attributes and their storage format leaving these details to the relational schema.

A relation, *reference*, is defined in the conceptual model. If an object A has a reference relationship with object B, then A cannot be compiled and executed without the definition of B. The following table shows all the meaningful reference relationships among five kinds of object in a C program.

A C language parser extracts these defined objects and relationships from a program, and stores the information in an INGRES database. Three major types of information retrieval are illustrated – 1) info: retrieval of attribute information of an object; 2) rel: retrieval of relationships between two object domains; and 3) view: view the definition of an object.

Table 5 Reference relationships among C programs

OBJECT KINDS #1	OBJECT KINDS #2	INTERPRETATION
File	File	File1 includes File2
Function	Function	Function1 refers to Function2
Function	Global Variable	Function refers to Global Variable
Function	Macro	Function refers to Macro
Function	Type	Function refers to Type
Global Variable	Function	Global Variable refers to Function
Global Variable	Global Variable	Global Variable1 refers to Global Variable2
Global Variable	Macro	Global Variable refers to Macro
Global Variable	Type	Global Variable refers to Type
Type	Type	Type1 refers to Type2
Type	Macro	Type refers to Macro

Source Code Algebra

Source Code Algebra (SCA) [PAU94] is an approach to set up a formal framework to support source code queries. In [PAU94], Paul and Prakash first argued that relational source code models are inadequate to formally model the complex structure of source code. In their paper they argue that relational algebra – the foundation of relational databases, cannot support a wide variety of atomic (e.g. *integer*, *string*) and composite data types (e.g. while-statement *has* condition and body, or statement-list is a *sequence* of statements) in source code. Besides, relational models also not able to characterize type hierarchy relations (e.g. for-statement *is kind of* loop-statement).

In their SCA approach they try to address these limitations of relational algebra. The approach is based on the generalized order-sorted algebra [BRU90], as the foundation for building a source code query system. Generalized order-sorted algebra is essentially a many-sorted algebra (in contrast to one-sorted relational algebra) with a partial order defined on its sorts. They consider SCA in being capable of modeling source code structures that the various source code data types (atomic and composite) are ordered by a subtype of relationship. SCA can sufficiently model source code information and contains very expressive operators for making a variety of queries of interest for software maintainers. However, it is still unclear if these SCA expressions can be evaluated efficiently.

Graph based approach

More recently, several graph based approaches (GUPRO [LAN01], CLG [KUL00], etc) have been proposed to overcome the limitations of the relational model [PAU94, KLI03]. These approaches commonly represent source code as graph structures (node as source code object and edge as relation), and use GReQL [KUL99] to query the graph. For a more detailed coverage, the reader is referred to [AN01, KUL00].

Limitations of Syntactical Models and Queries

Representing relational views of source code and using SQL-styled language to query them have some inherited problems: 1) relational model is difficult to capture the complex structure of source code structure, especially, the type hierarchy relations [PAU94]; 2) the expressiveness of SQL lacks the ability to express transitive closures [KLI03]; and 3) there will be noticeable performance problem extending relational query language.

Graph based approaches [LAN01, KUL99] overcome some of the problems of relational approaches. The graph query language GReQL [KUL99] supports transitive closures and enables the user to formulate regular path expressions. The major benefit of GReQL is the ability to directly traverse the graph, whereas in SQL different tables have to be joined many times. However, current graph based approaches are mostly memory based, and the graph query language cannot capture type hierarchy.

Some other approaches [MK98] use network structure to represent source code. For example, in Rigi [MK98], a special purpose semantic network data model is adopted to represent objects and relations in source code. A language (RCL) is designed to manipulate them. However, the problem of these approaches is the lack of query languages with well-defined semantics and queries have to be procedural.

2.1.2.5 Dependency analysis

The majority of existing reverse engineering and program comprehension analysis tools provide some type of dependency analysis at the source code level. In the context of substituting components at the source code level, the same dependency analysis techniques can be re-applied. Dependency analysis techniques provide (a) detailed insights of the low level relationships among different statements, variables, etc., supporting the general comprehension process (b) They can guide programmers in determining the scope, affect and type of the component substitution (in the context of substituting a set of statements or a function) with an equivalent component.

Static approaches tend to be necessarily conservative and include a large number of superfluous method invocations. Various static approaches exist that attempt to minimize the amount of irrelevant information. Although dynamic approaches are based on an exact trace of the methods that are invoked at run-time, they are unsuitable if we need to understand software without executing it (as is the case with inspections). Another challenge with respect to dynamic approaches is that they rely on the determination of a suitable set of test cases that are exhaustive and representative with respect to the use-cases. They also rely on the availability of a system that is executable in the first place. In object-oriented systems, it is particularly common that incomplete systems (such as frameworks) are created, which can then be used in different contexts.

One common approach to static dependency analysis is program slicing [Wei81]. Weiser 9Wei81] defined a slice *S* as a reduced, executable program obtained from a program by removing statements such that *S* replicates parts of the behavior of the program. Informally, a static program slice consists of those parts of a program that potentially could affect the value of a variable *v* at a point of interest. Canfora, Cimitile and de Lucia present in [CAN98] a pre/post condition slicing algorithm that is an extension Weiser's static slicing algorithm capturing a program's behavior by utilizing pre and post conditions to identify statements that can be removed. Conditioned slicing is a more general slicing method subsuming both static and dynamic slicing. Statements can be removed if they cannot lead to satisfaction of the negotiation of the post condition when executed in an initial state, which satisfies the pre-condition. Harman, et al present in [HAR97] an extension of this algorithm. Symbolic executions are then used to compute a condition slice.

Korel and Laski introduced in [KOR90] dynamic slicing that can be seen as a refinement of the static approach. A dynamic slice preserves the program behavior for a *specific* input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which a program terminates. Two major dynamic slicing approaches have evolved. Backward algorithms trace in reverse order a recorded execution trace to derive data and control dependencies that are then used for the computation of the dynamic slice. In contrast, forward algorithms overcome a major weakness of the backward approach - the necessity of recording the execution trace during program execution. Hybrid algorithms that combine static and dynamic information for the slice computation were introduced in.

Different slicing techniques and criteria are required because various applications require different properties of slices. Program slicing is not only used in software debugging but also in software maintenance and testing. Program slicing is an already established method for analyzing sequential programs. However, there are only a few existing slicing algorithms that support slicing of distributed programs.

2.1.2.6 Tracing

Dynamic analysis evaluates a software system or component based on their execution behavior and plays an important role in comprehending distributed systems. Dynamic analysis takes advantage of the more detailed and precise (compared to static analysis) information available based on some program inputs [BAL99]. Instrumentation and program tracing both are major approaches used to collect information required for dynamic analysis. Program tracing can be described as the process of recording program executions and has been applied in debugging, testing, monitoring, and comprehension, etc. According to the IEEE standard 610.12 [IEE90], a trace is a record of the execution of a computer program, showing sequences of instructions executed, names and values of variables, or both. Its types include execution trace, retrospective trace, subroutine trace, symbolic trace, and variable trace.

In order to obtain a statement level execution trace, users need to produce a record to show the exact statements that were executed and the execution order of the statements during a particular run. The original programs need to be instrumented at the statement level. When a statement is executed, the instrumented instructions are triggered, and generate a record for the statement. It is common that a trace, once generated, is stored in a file. A trace file contains

therefore a series of events where one is an execution of a statement. El-Ramly *et al.* [ELR02] propose a dynamic approach that records user interaction with the system. Based on these interactions, data mining and pattern matching techniques can be applied. Egyed [EGY01] proposes an approach that uses run-time information to produce traces between scenarios, model elements and the system. In this approach, the user supplies a series of representative test cases and an executable version of the system. The system is executed and a 'footprint graph' is constructed. This is used as a basis for automatically generating further traces.

When tracing a distributed system, users typically obtain separate trace records from various processes. Communication and synchronization information must be captured in order to match trace records from different processes. In message-passing based distributed systems, communication information is traced by recording each *send* and *receive* methods when they are invoked. A message-passing system is first instrumented at the method level to be able to trace *send* and *receive* methods. When the methods are called, the instrumented code will generate communication records. Logical clocks or timestamps are typically used to accomplish synchronization between processes. When a communication record is generated, a timestamp label is attached to the record. According to the timestamps of all communication records, these records can be ordered and connected between a *send* record and its corresponding *receive* record. Since extra instructions have to be instrumented to record the communication and synchronization information in a distributed system, tracing a distributed system is much more difficult than tracing a traditional sequential program. Moreover, extra instructions mean that tracing a distributed system will result in an additional overhead compared to tracing a sequential program. It has to be noted that instrumentation of the source and bytecode may modify the program behavior, and the non-deterministic behavior of these systems can cause situations where the recorded traces will not correspond to the non-instrumented program executions.

One of the main challenges of tracing realistic sized programs is to resolve the size explosion problem. A long execution may generate a huge trace file, and creating these huge traces leads to additional overheads. Furthermore, operating on a huge trace file is inefficient and time-consuming. Hamou-Lhadj and Lethbridge [HAM01] classify the techniques used to reduce the amount of trace information into two categories, trace exploration and trace compression. The first one is concerned with the ability to browse the content of trace efficiently, and the second one directly focuses on reducing the size of the trace, by removing or hiding some of its components. They summarize that the data collection techniques, pattern matching, sampling, hiding components and filtering are used in tools to reduce the size of traces.

Instrumentation

Instrumentation typically modifies either source code or binary code with additional monitoring instructions to allow for the collection of run-time states and information of a system. The collected information can then be utilized by additional dynamic analysis tools, such as profilers, instruction trace generators, monitors, test tools, etc. For Java applications, there are three types of instrumentation techniques. One approach is to instrument at the source code level. The second approach instruments the bytecode generated for an existing system. The third approach is to interface with the Java Virtual Machine through the Java Debugger Interface (JDI). The third approach is not covered in this research due to its limited applicability. Interfacing with the virtual machine can provide detailed run-time information, however the performance penalty using the JDI is so significant that its applicability is limited only for very short program executions or collection of high-level information. Depending on the level of analysis (variable, statement, function or class level), the overhead associated for extracting run-time information using the JDI is between 10 – 1000 times of the original execution time.

During source code instrumentation, extra instructions are inserted directly into the original source code. After the instrumentation, the modified source code has to be recompiled in order to be able to execute the instrumented application and obtain dynamic information. Therefore, the source code of an application must be available and moreover, there is an additional overhead for recompiling the instrumented source code. On the other hand, the first advantage of the source code instrumentation is that no specialized runtime environment is required. After the instrumented source code is recompiled, it can run within the same program environments as the original program, i.e., the same JVM and class loader. The second advantage of source code instrumentation is its support for statement level source code analysis, such as source code coverage tools, statements and branch coverage [Clo05]. The statement level information is available for the source code instrumentation, because it operates directly on the original statements of source code.

2.1.2.7 Statement level Metrics

Fenton [FEN99] described software engineering as a "collection of techniques that apply an engineering approach to the construction and support of software products." Software metrics were introduced to support the most critical issues in software development and provide support for planning, predicting, monitoring, controlling, and evaluating the quality of both software products and processes. Many concepts have been presented to evaluate the process of software development and the quality of software design. Software measures enable the early identification of maintenance and reuse issues in existing systems. It has been shown that software metrics can provide software engineers and maintainers with guidance in analyzing the quality of their design and code, and its possible maintainability and comprehension [FEN99].

There exists a number of source code (statement level) based metrics, that can not only be applied to evaluate the quality of a software system, but also its maintainability, reuse and comprehensibility. In the following section, some of these metrics are briefly described. The presented metrics also cab provide guidance in predicting the complexity of the component substitutions and can be applied as an indicator to improve the quality of other techniques, like the coverage metrics can be used to evaluate the quality of profiles and therefore the quality of the collected dynamic data.

2.1.2.7.1 Cyclomatic complexity

Cylcomatic complexity is a well established software metric (measurement) concept that originated in the computational complexity theory. It was developed by McCabe and is used to generally measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. The cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity may be considered a broad measure of soundness and confidence for a program. Cyclomatic complexity is often referred to simply as program complexity or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format. Applications include being a measurement for maintainability. It might also indicate places in the source code where some of the major business logic is represented.

2.1.2.7.2 Coverage metrics

Code coverage analysis is a structural testing technique (a.k.a. glass box testing and white box testing). Structural testing compares test program behavior against the apparent intention of the source code. This contrasts with functional testing (also known as black-box testing), which compares test program behavior against a requirements specification. Structural testing examines how the program works, taking into account possible pitfalls in the structure and logic. Functional testing examines what the program accomplishes, without regard to how it works internally. Structural testing is also called path testing, since one chooses test cases that cause paths to be taken through the structure of the program. Statement coverage is achieved when every executable statement in the program is invoked at least once. Achieving statement coverage shows that all code statements are reachable, reachable based on test cases developed from the requirements).

2.1.2.7.3 Condition/Decision Coverage

Decision coverage requires two test cases: one for a *true* outcome and another for a *false* outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. But, not all decisions are simple. Condition coverage requires that each condition in a decision takes on all possible outcomes at least once (to overcome the problem in the previous example), but does not require that the decision take on all possible outcomes at least once. In this case, for the decision (**A *or* B**) test cases (*TF*) and (*FT*) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two tests cases is required for each decision.

Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between *true* and *false* and to toggle each condition value between *true* and *false*. Hence, a minimum of two test cases are necessary for each decision. Using the example (**A *or* B**), test cases (*TT*) and (*FF*) would meet the coverage requirement. However,

these two tests do not distinguish the correct expression (**A *or* B**) from the expression **A** or from the expression **B** or from the expression (**A *and* B**).

2.1.2.7.4 Identifier Complexity

Current software inspection techniques are driven by inspecting and understanding all code and its dependencies. This is an expensive, time consuming, often unrealistic approach, particularly for very large systems and their executions. An "as-needed" reading approach has to be adopted to deal with the possibly large amounts of de-localized information. The aim of any inspection approach has to be to limit the number of dependencies that have to be analyzed and comprehended. Measuring complexity reflects attributes of human cognitive comprehension characteristics. Software defects are often the result of the incomplete or incorrect comprehension of a program segment [FEN99]. Short term memory can be thought of as a container, where a small, finite number of concepts can be stored. If data are presented in such a way that too many concepts must be associated in order to make a correct decision then the risk of error increases. Therefore, the location of a program segment that presents a comprehension challenge to the software developer can be the basis for isolating code at a greater risk of defects and therefore guide the inspection of existing source code for the purpose of refactoring, restructuring and preventive maintenance.

When a programmer encounters an unfamiliar code segment, the language component of the code is not new. The use of identifiers such as variables and method labels must be carefully observed during the comprehension of a software system [WEI81]. An indicator of how much tracing is involved in understanding a program is the external coupling, or references to external blocks of code. In computer programs, identifiers represent concepts. If the program is unfamiliar to the programmer, then the identifier density is a good predictor of error, provided time is constrained. In order to adjust for individual differences that experience brings, a strategy can be applied given our understanding of human thinking. In program code, some terms belong to the programming language and others are terms defined within the program. An experienced programmer recognizes keywords and understands their meaning without the need to search for their definition. Variables, classes, methods and other programmer-defined labels must be traced to identify their definition and application. They are generally unfamiliar to the programmer who must store them in short term memory in order to understand their role in the code. Identifier density can be defined as all occurrences of programmer defined labels divided by lines of code.

Some other more traditional complexity metrics can be supported by the fact that they are clearly related to cognitive limitations. These include lines of code, fan-out, and decision points such as McCabe's cyclomatic complexity. When a program segment has a high number for LOC, it will have enough features, such as identifiers, that it becomes difficult to understand correctly. When a program segment has a high fan-out, the time spent tracing references can increase and comprehension is compromised. Finally, when there are many possible paths to be taken within a module as when McCabe's cyclomatic complexity would get a high value, again tracing becomes demanding and it can be difficult to correctly interpret.

2.1.2.8 Discussion – FOSS component substitution at the statement level

Information needs: All of the presented techniques provide very detailed information of the source code to be replaced/substituted. Due to the semantics and syntax of source code, there is a need for parsers and fact extractors to be customized to the specific environments (programming language). Furthermore, most of the presented techniques require that the source code is based on only *one* single programming language rather than an implementation utilizing a set of different programming and scripting languages. This single language assumption is one of the major limitations for these approaches, restricting their applicability for component substitution in a mixed programming environment as it is often found in FOSS systems. Similar programming environment issues can be found for the tracing and profiling tools, due to the fact that also these tools require some form of parsing/fact extraction.

Techniques: The techniques review in this section can be classified in three categories. Fact extractors and parsers are the foundation for almost any of the presented source code analysis techniques. The extracted information is stored in a program model for permanency and later information retrieval, in the form of an XML file, relational database or object-oriented database. The challenge in particular with parsers is that they are typically programming

language, even version specific and their precision, the level of detail will affect the use of the extracted information. The most commonly used comprehension techniques by software maintainers are based on browsing and querying the source code. From a component substitution view point, these parsing and fact extracting techniques and the program model used to store the information are probably one of the more relevant sources for component substitution at any of the abstraction levels.

Similarly, profiling and tracing tools provide an important foundation to support both component substitutions at the statement level, as well as the substitution process at higher levels of granularity. The challenges associated with profiling and instrumentation tools are similar to the parsing and extraction tools. Typically, these tools are specialized to support only a specific programming language. Other challenges include modifications to the program behavior, leading sometimes to non-deterministic behavior of the programs that are monitored. Also, scalability is an important issue that has to be considered, especially for larger applications, or long running applications.

Lexical and syntactical query approaches have difficulties in exploring domain semantics conveyed in source code. For example, in order to find a variable that represents "current temperature", maintainers have to use regular expressions provided by lexical tools or LIKE clause in SQL to find clues, which is opportunistic and inaccurate. This inaccuracy and lack of insight might limit the support they can provide during the component substitution process.

The use of statement level based metrics during component substitution can provide some support for profiling (coverage metrics) by improving the coverage and quality of the traces, as well as providing some guidance with respect to the comprehension difficulty level involved in understanding the source code (complexity metrics).

Information provided: The information provided by these techniques is mainly focusing on providing and populating initially a static program model that can be further utilized either by dependency analysis (e.g. program slicing) techniques or querying (lexical and semantic) approaches. The tracing and monitoring of information provide the basis for dynamic analysis (e.g. program slicing) and other analysis techniques at higher level of abstraction that can support the component substitution process.

Applicability for component substitution: The challenge of these statement level techniques is that their level of granularity is in general very fine and detailed, leading typically to a large amount of information that has to be stored, analyzed and interpreted. However, this detailed statement level information provides essential support for the comprehension of the source code at the statement level and its internal dependencies.

Table 6 Applicability of analysis techniques at the statement level

Technique	Application
Parsing, fact extraction, lexical analysis, queries, token analysis, Strings and token matching	Lexical analysis, queries, fact extraction flow analysis, token matching Query support (semantic, lexical) Basis for most comprehension and reverse engineering tools
Dependency analysis, semantic analysis, slicing, impact analysis	Dependency analysis, slicing, flow analysis, impact analysis
Program model	Basis for most reverse engineering tools
Tracing, profiling, monitoring	Dynamic flow and dependency analysis, profiling
Statement metrics	Comprehensibility, profiling, maintainability

2.2 Function/Class level component substitution

2.2.1 COTS

From a binary perspective, there exist only view techniques that are specific to identifying, comprehending and substituting components at the function/class level. This level of abstraction is typically covered by the statement level techniques introduced in section 2.1.1 of this report, including tracing, binary reverse engineering and binary dependency analysis. In particular, the collected traces can be abstracted to the level of message passing, function calls. This information can either be visualized using traditional UML diagram techniques (e.g. collaboration diagrams, sequence diagrams) or the information can be analyzed using some of the traditional coupling metrics, which are similar to the ones discussed in the next section (FOSS – function/class level approach). The challenges for the trace collection are manifold, but they can be summarized as follows. (1) Identification of adequate test cases that reflect real world usage. (2) Scenarios of the system, the collection, storage and retrieval of the traces and the overhead associated with them. (3) The interpretation and analysis of the large amount of data.

Table 7 Overview COTS analysis at the function/class level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Decompilation	Bytecode	Assembly code, Java code (decompiled)	Flow analysis Dependency analysis	Y	N
Flow analysis Dependency analysis (mainly control flow), slicing	Bytecode Decompiler (Assembly or Java Code)	Control and data flow analysis, limited slice	Flow Analysis Dependency analysis	Y	Y
Instrumentation (Virtual machine), bytecode	Bytecode	Execution traces, call sequence, metrics	Dynamic Flow and Dependency analysis Call dependencies	N	Y

2.2.2 FOSS

With the availability of source code, there is a full range of techniques and tools available that can support the component substitution process at the function and/or class level. In what follows, an overview of these techniques, as well as their information needs and the information they provide towards a component substitution process are introduced and discussed. Techniques at this level mainly focus on identifying structural dependencies among functions and/or classes (both static and dynamic). Most of the presented techniques are based on some type of underlying program models that are queried to extract structural information from the underlying source code for further processing. It has to be noted that these techniques at this abstraction level cannot be seen in isolation. They both depend on information (resources) provided by more fine grained techniques (e.g. parsers, program models) as well as they serve as input to some of the techniques at higher levels of abstractions. Due to the limited scope, these techniques might not be sufficient on their own to support a successful component substitution. Additional information derived from other techniques (at different abstraction levels) will have to be used to complement the approaches surveyed in this section.

Table 8 Overview FOSS analysis at the function/class level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Design pattern recovery	Source code, program model language parser, pattern description	Identified design patterns	Comprehension, grouping, domain knowledge	Y	Y
Metrics	AST, program model, call dependencies	Different coupling, cohesion measurements	Maintainability, impact analysis	Y	Y
Slicing	Program model, source code, AST, control and dependency graph	Slice (executable, non-executable)	Comprehension, impact analysis, testing	Y	Y
Visualization	AST, program model	Abstract views, UML class model, sequence diagram.	Comprehension	Y	N/Y
Dependency Analysis	Source code, program model, control and data flow	Call-graph, call dependencies	Grouping, interaction	Y	Y

2.2.2.1 Dependency Analysis

Static Approaches

The major static approaches at this abstraction level focus on some type of call graph/message dependency graph. Di Lucca *et al.* [14] propose an approach that is based on the premise that a scenario starts with a system-level input and ends with a system-level output. They represent the message sequences in the form of a Method-Message Graph (MMG). 'Threads' of message invocations are extracted from the graph and collated to form use-cases. Tonella and Potrich [22] provide a reverse-engineering approach for interaction diagrams from C++ code. Acknowledging that a purely static approach is over conservative, they use two mechanisms called partial analysis and focusing to ensure that the average size of a graph is small enough to be of use. They validate their approach by applying it to a substantial real-world project. Qin *et al.* [15] propose an approach based on constructing a call graph-based abstract representation of the subject program called the Branch-Reserving Call Graph (BRCG). These represent calls between methods and retain control dependence information, so that predicate statements that control the execution of a given procedure call are integrated. Because no prior use-case information is used and the approach is static, it returns all possible execution scenarios of the system. This can be alleviated by pruning nodes using a graph-based importance metric [WAL04].

Dynamic Approaches

As discussed in the previous section, dynamic analysis can be very helpful for understanding the behavior of a large system. Understanding an object-oriented system, for example, can be very hard if one relies solely on the

source code and static analysis. Due to the abundant use of polymorphism in object-oriented software, the relationships between the system artifacts tend to be obscure. Dynamic data gathered from the execution of a scenario of the program is typically stored in so-called execution traces. These execution traces tend to explode in size, as the amount of dynamic data collected during the execution of a simple scenario is huge. Presenting this data to the user, without any form of further processing, might provide meaningless results. What are needed are techniques that improve the intelligibility of the trace. In other words: a huge trace has to be chunked in readable and understandable parts. Preferably, these chunks presented to the user are essential in building up the knowledge of the software project under study. Existing solutions rely on visualization schemes for presenting the trace to the user in a readable and understandable way. The amount of data however, remains the same. As such, it is up to the user to decide which information is appropriate and which information he/she actually needs. Another approach in this context is applying heuristics to the event trace in order to find either (1) what parts of the trace can be removed without losing information (e.g. duplicate parts) or (2) what parts of the trace are absolutely necessary in order to extract moderately sized high-level representations from it.

Traces also form the basis for various dynamic dependency analysis techniques. El-Ramly *et al.* [ELR02] propose a dynamic approach that records user interaction with the system. Based on these interactions, data mining and pattern matching techniques can be applied. Egyed [EGY01] proposes an approach that uses run-time information to produce traces between scenarios, model elements and the system. In this approach, the user supplies a series of representative test cases and an executable version of the system. The system is executed and a 'footprint graph' is constructed. This is used as a basis for automatically generating further traces. Among other analysis techniques, one can identify dynamic dependency graph based approaches, like program slicing. The approach is based on system dependency graphs, which are extended versions of a traditional program dependency graphs, that includes not only control and data dependencies but also call dependencies. The comprehension of a distributed program in particular remains a major challenge due to the timing related interdependencies among processes that add complexity to the comprehension process (compared to traditional sequential programs).. Due to the majority of systems these days being distributed applications, the analysis and comprehension of these distributed systems becomes a major challenge. From a component substitution perspective, comprehending the program behavior becomes an important issues. Program slicing is one technique that can provide additional insights when applied at the function and class level to analyze and comprehend the interaction and communication that occurs within distributed programs.

2.2.2.1.1 Program Slicing of Distributed Programs

The notion of program slicing [WEI81] was extended in the last decade, in attempt to provide support with new technologies and programming paradigms. These extensions include support for object-oriented and distributed programs. There exist several surveys on program slicing techniques and on the applicability of slicing in comprehending object-oriented software systems. The reader is referred to existing surveys covering this topic in detail [BIN04]. The comprehension of a distributed program in particular is a major challenge due to the timing related interdependencies among processes that add complexity to the comprehension process (compared to traditional sequential programs). Program slicing can significantly reduce this comprehension effort, by eliminating portions of the program and thereby providing some level of abstraction (with respect to a particular slicing criterion) of a program and its execution. The following is an overview and discussion of existing static and dynamic slicing algorithms for distributed programs.

Static slicing of concurrent programs [RIL03]

Cheng presents in [CHE93] an approach to static and dynamic slicing of concurrent programs based on *program dependence nets (PDN)*. The slices computed by Cheng's algorithm are not precise, as they do not consider dependencies between parallel-executed statements that are not transitive. As Tip notes [TIP93], Cheng does not state or prove any property of the slice computed by his algorithm. Zhao [ZHA93] addresses static slicing of Java programs using thread dependence graphs (TDGs). This algorithm is an extension of Cheng's earlier work [CHE93] for object-oriented language constructs and provides support for method calls and synchronized methods, but not for synchronized statements. Zhao also presents an algorithm for the computation of executable slices for concurrent logic programs on the argument level. However, the paper does not provide formal semantics to represent the communication dependencies and the algorithm does not always compute executable slices. Krinke [KRI01] considers in his work static slicing of multi-threaded programs with shared variables and focuses on issues associated with interference dependence. In this approach however, there is no explicit support for a synchronization

mechanism. Mueller-Olm and Seidl provide in [MUE01] a discussion on the computation of optimal static slices for parallel programs. They show that the computation of minimal static slices for parallel programs is not decidable which has also been shown by Weiser [WEI81]. Static slicing algorithms for distributed programs suffer from their limited ability in handling and analyzing communication dependencies that often lead to larger slices than necessary and limit their usability for larger software systems.

Dynamic slicing of concurrent programs

Korel and Ferguson [KOR92] extended Korel's original dynamic slicing algorithm [KOR87] for distributed programs with Ada-type rendezvous communication. For a distributed program, the execution history is recorded as a distributed program path. The slice is only guaranteed to preserve the behavior of the program if the rendezvous in the slice occurs in the same relative order as in the program. Duesterwald et al. present in [DUE92] a hybrid parallel algorithm to compute dynamic slices for distributed programs using a distributed program dependency graph. The algorithm combines both static and dynamic information to compute a slice. Additionally, they propose to transform non-deterministic communication constructs into deterministic ones to provide executable slices. However, their approach requires the user to specify a slicing criterion for a particular process and execution position and it does not consider timing issues. Kamkar and Krajina introduced in a dynamic slicing algorithm for distributed programs that utilize a distributed dynamic dependency graph to compute a dynamic slice. The algorithms presented by Kamkar, Duesterwald and Korel [DUE92, KOR92] only support a subset of a structured programming language and do not explicitly consider any timing related issues. Smith and Korel present a dynamic slicing algorithm that focuses on the slicing of even traces, allowing for a reduction of the required execution trace that has to be recorded. Garg and Mittal present in [GAR01] the notion of a slice of run, which captures only those consistent global states of the original computation that satisfy a global predicate. In that case, attention is focused on identifying (and possibly visiting) exactly those states of a run in which the global predicate holds. The motivation and application domain for the algorithm presented by Garg and Mittal differs from traditional program slicing by focusing on the reduction of the state lattice size and therefore, the search space, rather than identifying or removing irrelevant statements from the program.

2.2.2.1.2 Design pattern recovery

One of the common ways of understanding software is to develop a global picture of the system and its internal structure by representing this structure using for example UML class or ER diagrams. Both of them reflect the physical and logical structure of the classes and their functions as well as their communications. However, this is information is often still insufficient for a maintainer to fully comprehend the *purpose* of a given piece of code. One of the possible reasons is that existing design notations focus on communication the *what* of the design, but almost completely ignore the *why*. Design patterns are design elements that capture "the rationale behind recurring proven design solutions and illuminate the trade-offs that are inherent in almost any solution to a non-trivial design problem" [GAM94]. A design pattern, as a high-level design element, describes a commonly-recurring structure of communication components that solves a general design problem within a particular context. The description of a pattern contains not only the knowledge about the components and the inter-relationships, but also the alternatives and design decisions behind the design and implementation. Knowing the patterns existing in the legacy system will give additional information to the reader, and thus much improve the efficiency of software understanding. In this point of view, we consider recovering design patterns in legacy systems an important step in the software comprehension process, which is, partially answering the question – "why the software designed like that". This additional insight therefore not only provides additional insight about the design structure, but also allows to infer certain domain expertise, that can support the component substitution process. The benefits of identifying design patterns can be highlighted as the following:

- Comprehension – recovering design patterns in legacy systems and building a pattern-level representation of the system to facilitate software understanding. Furthermore, incorporating domain knowledge and other software artifacts to provide guidance in rediscovering the design and understanding of the design decisions and rationales beyond the raw source code.
- Re-documentation –pattern languages can be used to document the design of the legacy system that will significantly improve the precision and concision of theses documents.
- Refactoring – both identified design patterns in an existing system and their relationships may improve the quality of source code, by allowing for refactoring of parts of the source that were not well designed.

- Validation – A design pattern correspond to a proven solution with respect to a particular problem. The recovery of the implemented patterns can provide some additional insights of the problems the software is trying to address or solve. Furthermore, we can use pattern recovering tools to validate whether the current design/solution fully satisfies the requirement of the pattern implementation by verifying if a certain pattern can be identified.

The recovery of design patterns can facilitate and support reverse engineering tools by improve the expressiveness of the created representations, and thus help people to understand the design decisions behind the source code. One approach to design pattern recovery is presented in [ ], where both design patterns and source code are described in PROLOG and the PROLOG engine can be used to do the matching job. The recovering process starts from representing each pattern as a static OMT diagram and converting the diagram to a PROLOG representation, which is one rule for each pattern. The C++ header files are analyzed and the information is stored in the repository in OMT form, which will then be translated into to PROLOG representation. At last, a PROLOG query will detect all instances of design patterns from the two repositories. The collected information is just based on header files, without semantic analysis in method body limiting its applicability to a very few structural patterns, like Adapter, Bridge, Composite, Decorator, and Proxy.

Other approaches to recovering design patterns from source code are mainly based on a multi-stage reduction strategy, using software metrics and structural properties to extract structural patterns from OO design or code. First, an Abstract Object Language (AOL) representation is created. In a second step, the AOL representation is used to generate an AOL abstract syntax tree. Then relevant class metrics are extracted. Finally, pattern constraints like metrics constraints, structural constraints, and delegation constraints are applied to the AOL abstract syntax tree and class metrics obtained in the previous steps and a set of pattern candidates are conducted. Yet another approach for discovering design patterns is a metrics tool that analyzes the quality of a software architecture given as a set of UML diagrams by transforming the C++ source code into UML class diagrams. These diagrams are then traversed and matched against a set of predefined design patterns that consists constraints formulas to describe each UML class.

A common challenge for all of these design pattern recovery techniques is their accuracy in identifying design patterns and the type of design patterns supported by these techniques (structural or behavioral patterns). It should also be noted that most of these existing techniques are limited to the recovery of structural patterns in the source code and that these techniques only support typically a fixed set of predefined patterns.

2.2.2.1.3 Metrics

Constantine and Yourdon defined coupling based metrics on the relationship of subroutines as measurements for procedural systems. In the context of OO programs, the minimization of connections between classes or methods also minimizes the paths along which changes and errors can propagate into other parts of the system. Strong coupling complicates a system, since a module is harder to understand, change, or correct by itself if it is highly interrelated to other modules. Hence, to reduce the complexity of a system, it is necessary to keep the weakest possible coupling between classes. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. Page-Jones [PAG80] gave three principle reasons why low coupling between modules is desirable:

(1) it reduces the chance that a fault in one module will cause a failure in other modules,
(2) it improves modularity and promotes encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore, maintenance is more difficult.
(3) it reduces programmer time to understand the details of other modules.

For the same reasons, metrics should be applied during component substitution. Metrics have the advantage of being relatively easy to compute. They can, as mentioned before, provide a general indication on the effort required to perform a substitution at the class or function level. In particular, coupling measurements can be used to support the comprehension and analysis process (e.g. ripples effect analysis, change impact analysis, etc.). Briand et al. describe coupling as the degree of interdependence among the components of a software system and they present a unified framework for coupling measurement in OO system [Bri97, Bri98].

A large number of coupling measurements for object-oriented programs have been introduced and discussed in the literature. The most widely used and criticized measurement suite is from Chidamber and Kemerer [CHI91], where they identified 6 object-oriented measurements: Weight Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Lack of Cohesion in Methods (LCOM). These were proposed with the intention to be independent of OO language implementation details. Li and Henry[GEN01] assessed Chidamber and Kemerer[CHI93] measurement suite and introduced message passing coupling (MPC) data abstraction coupling (DAC), and the number of local methods (NOM).

MPC

Li and Henry [HEN01] examined the original measurement suite of Chidamber and Kemerer and proposed a new measurement Message Passing Coupling (MPC), which is used to measure the complexity of message passing among classes. Message passing is one of the typical types of communication between the objects in the object-oriented paradigm. When an object needs some service that another object provides, messages are sent from one object to the other. A message is usually composed of the object-ID, the service (method) requested, and the parameter list for that method. MPC is the number of send statements defined in a class, where a send statement is a message sent out from a method of class A to class B. Although messages are passed among objects, the types of messages passed are defined in classes. Therefore message passing is calculated at the class level instead of the object level [GEN901].

RFC

Response for a class (RFC) measures the size of the response set of a class [FEN97]. The response set of a class consists of the set M of methods of the class, and the set of methods invoked by methods in M. In other words, the response set is the set of methods that can potentially be executed in response to a message received by an object of that class [CHI91,]. RFC measures the communication object among classes through message passing. A message can cause an object to “behave” in a particular manner by invoking a particular method or set of methods. Methods can be viewed as definitions of responses to possible messages. It is reasonable therefore, to define a response set for an object in the following manner: Response set of an object = {methods that can be invoked in response to a message to the object}. Note that this set will include methods outside the object as well, since methods within the object may call methods from other objects.

2.2.2.2 Visualization (static and dynamic)

2.2.2.2.1 Visualizing Class Interfaces with Formal Concept Analysis

Source code views are primarily based on some diagrammatic notations that mostly have been evolving from the early days of computing. However, for large and complex software systems, the comprehension of such diagrammatic depictions is restricted by the resolution limits of the visual medium (2D computer screen) and the limits of a user’s cognitive and perceptual capacities. These limitations are based on the following observations:

(1) Scalability problems: For large systems, the resulting visual representations tend to be cluttered, often resulting in information overload problems.
(2) Layout problems: The awkward and sometimes arbitrary layout techniques in use tend to focus on minimizing line crossing and often obscure important software aspects such as patterns and other types of relationships.
(3) Non-intuitive navigation: Typical navigation includes pan-zoom and overlapping of multiple windows. There are some visualization mappings such as fisheye-views, perspective information walls, and hyperbolic trees etc. which offer some solutions for focus versus detail. They assist the user in not getting lost in the visual space, but even these are limited in their scalability for large software.

One of the major challenges for visual representations to be applicable during a component substitution process is the need, not only to provide a visual abstraction, but also to allow for a meaningful interpretation of the data to be visualized. Combing visual representation with analysis techniques will improve the interpretation of the visualized data and allow for meaningful grouping and filtering of the information.

2.2.2.2.2 Filtering

As Ben Shneiderman described, filtering is an important intermediate step in reducing the information volume that has to be displayed. Two major categories of filtering techniques can be distinguished - structural and semantic. The most commonly used structural filtering techniques in software visualization are based on creating hierarchical views of a software system. Information reduction by structural filtering techniques is one approach to improve the scalability and applicability of visualization techniques. However, without a meaningful interpretation, these filtering techniques might sill not be sufficient in interpreting the systems. Source code analysis techniques can be applied to filter and group the analyzed source code to provide for more meaningful interpretation of the visual representations. One well-known approach to source code analysis is program slicing [WEI81, BIN04, and KOR87].

2.2.2.2.3 Grouping and Layout

Filtering techniques are only a step towards improving software visualization. They are not sufficient to close the conceptual gap between representations created during the traditional forward engineering and the ones created as part of reverse engineering. The success of a visualization technique depends also on the visual organization and decomposition of the system. It has been shown that grouping, clustering and layout can improve readability by supporting representations that closely related to the mental model a programmer forms of a system during typical comprehension tasks [CHU05]. Traditional layout algorithms focus mostly on improving the readability by reducing the number of edge crossings in the diagrams [8]. One of the major shortcomings of this is that the only aspect considered is line crossing and other issues related to map the visual representation to the programmer's mental model are ignored. Many factors influence the original layout decision based on functional, domain knowledge, architectural aspects or just pure personal preferences that cannot easily, if at all, be extracted from the source code. The major objective of a good layout and/or grouping algorithm has to be the ability to recreate a representation that (1) closely matches the perception a user has of the system (either based on previous involvement with the system or based on existing domain knowledge), and (2) provide meaningful interpretations (views) of the system which might not correspond to a typical metric based layout/grouping approach. However, it has to be noted that recreating a graph layout that matches the original layout created by the designer/developer is a near impossible task.

Typical clustering approaches in software visualization are based on a hierarchical clustering or metrics based approaches. For large systems, the problem of minimizing the metrics becomes a greater challenge [4]. Not only does the complexity increase when trying to find an optimum solution, it also becomes often impossible to minimize the overlapping and crossing of edges. This quickly leads to visuals that are very difficult to comprehend. The major challenge of visual clustering and grouping is one of reducing visual aliases. Depending on the number of objects, clustering priority must be shared between user interpretation and clearer visualization. For the forward engineering of new systems, software visualization has been established as the tool of choice to control their complexity. Typically, UML diagrams are used as the graphical view to represent static and dynamic aspects of a software system []. However, the combination of applying both hierarchical and non-hierarchical relations poses a special challenge to a graph layout tool.

Design pattern recognition and visualization.

A significant effort has been devoted to recovering design patterns by focusing on the use of static information such as source code, while some works also include tracing dynamic code execution to detect behavioral patterns. Source code analysis can be used to detect design patterns and group the participating classes in a 3D representation to simplify diagram pattern matching. Furthermore, those design patterns have specific layouts that are familiar to most programmers, as suggested in. It is perceived that the comprehension process can be improved if the classes were grouped in the same or similar layout. This is due to several factors. One can understand the classes' roles due to their roles in the design pattern (which is known and documented in the existing literature). Therefore, the comprehension process can be moved to a higher level of abstraction, enabling visual pattern matching and their interpretations. This type of pattern matching is similar to recognition a stellar constellation (pattern) as a whole instead of individual stars.

2.2.2.3 Discussion – FOSS component substitution at the function/class level

From a pragmatic point of view, it is often difficult to classify a specific technique to be applicable to a certain abstraction level. Typically, these techniques can be applied to support component substitution at various abstraction levels. The following section briefly summarizes the findings of the survey and discusses some of the results

Information needs:

The general information needs of the presented techniques to support FOSS component substitution at the function/class level can be summarized as follows:

- Some type of a persistent and populated program model to extract dependency information.
- Execution traces at the statement, function and class level to support dynamic analysis techniques.
- Data and control dependencies for program slicing based approaches.
- Definition of design patterns to allow the recovery of design patterns from the program model.

Techniques:

The techniques reviewed in this section can be classified in three categories. Within each of these categories, one can distinguish static and dynamic approaches.

- Dependency analysis approaches that focus on intra procedural dependencies (call dependencies at the function or class level). The information can be further enriched by utilizing data and control dependencies to compute program slices. Also, some of the query approaches based on semantic analysis can be re-applied at this abstraction level to filter and analyze functional and class level dependencies. Another dependency analysis approach is the recovery of design patterns from the underlying program model. Common to all of these dependency analysis techniques is that they provide some detailed insights analysis to support the component substitution process at the function/class level.
- Metrics can provide an initial overview of the difficulty in comprehending the system and a general prediction of the effort involved in performing the component substitution. In particular, coupling metrics have been shown to provide some guidance during maintenance efforts.
- Software Visualization: Software visualization is gaining importance as a technique to abstract detailed information and to allow maintainers to gain an understanding of structural and dynamic aspects of a system. For function/class level substitution, visual representations either as static diagrams (class model, call graphs) or dynamic representations (sequence and collaboration diagrams) to represent the behavioral aspects, are commonly used. These diagrammatic notations however have to be further enriched through additional dependency and metrics analysis to help closing the conceptual gap between a maintainer's mental model and the visuals representations provided.

Information provided and applicability to component substitution:

The information provided by these techniques is focusing on some type of dependency analysis at the function/class level. Both the visual representations and the metrics information do not provide any detailed guidance in performing the component substitution. The information provided by the dependency analysis techniques themselves, like slicing, will generate a program or a system dependency graph, which include data, control and call dependencies. The insights gained from these dependency analysis techniques can provide maintainers with additional guidance during the substitution process, by supporting the comprehension process and interpreting the source code structures.

Table 9 Applicability of FOSS analysis at the function/class level

Technique	Application
Design pattern recovery	Comprehension, grouping, inferring domain knowledge
Metrics	Maintainability, identification of substitution complexity, quality evaluation, grouping for visualization techniques
Slicing, dependency analysis	Comprehension, guidance during substitution by focusing on relevant parts, grouping
Visualization	Comprehension, abstract representation of detailed information

2.3 Feature/Package level

The focus in the section is on the component substitution at the feature and/or package level and supporting techniques. This abstraction level corresponds most closely to the typical definition of a component. With an increase of the abstraction level, the information needs are also changing accordingly. At the feature/package level substitutions require less low level details about the source code and its structural aspects (call dependencies) become only part of the information needed to support a component substitution. There is an additional need to infer additional meaning that is more specific to the context and functionality of the system under investigation. At the feature level higher level analysis in the form of domain analysis and domain engineering techniques become a major factor. Typical analysis approaches at this abstraction level therefore focus on extracting and recovering of features and other domain knowledge from existing systems.

2.3.1 COTS

From both a system developer as well as from a software maintainer perspective, it is important that one understands the maintenance implications of integrating large numbers of COTS products into a system or substituting existing components with new components. In addition to the effort involved in identifying the scope of components, one has also to consider the potential impact of the modification on the existing system and the integration challenge associated with the component substitution. It is a known fact that different products (components) evolve differently, depending on the discretion of vendors. One has to expect a considerable effort involved in handling the continuous evolution of these components (e.g., understanding the impact of an upgrade on the rest of the system, making changes to glue code).

Table 10. Overview COTS analysis at the feature/class/package level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Protocol recovery	Static and Dynamic traces, profiler, domain knowledge,	Dependency analysis, Communication dependencies	Comprehension, impact analysis	Y	Y
Protocol validation	Static and dynamic traces, automata representation	Verification	Validating/verification that component protocols match before/after substitution	Y	Y
Wrappers/Glue Code	Component, interface	Standard interface, Access and Integration wrapper	Adaptation of legacy system, hooks, minimize change impact facilitates information and data exchange	Y	N
Component tailoring	Scripting, macro language	Plug-in, etc	Component enhancement	Y	N
GUI Ripping	GUI, executable COTS, test cases	GUI hierarchy	Reverse engineering of COTS, adapting legacy applications to new systems, Feature extraction	N	Y
Component metrics	Component, interface, function parameters	Meta information, component customizability, parameter completeness	Prediction of maintainability and substitution	Y	N
Compositional Component Adaptation	Standardized interfaces, dynamic information	Abstraction, separation of concerns	Maintainability, comprehension	Y	Y

2.3.1.1 Protocol Recovery [KOS02]

One of the major challenges in component substitution at the feature/component level for COTS is to identify the dependencies and communication among the different components in a system. Although research has answered the question on how protocols (limited to sequencing constraints that can be described by finite state automata) can be validated, there is basically no research conducted on how to identify these protocols if the original programmer did not specify them. Principally, there are three different sources of information in order to find hints on the actual protocol for an undocumented component.

Static and Dynamic Traces

The actual use of a component may be derived by dynamic or static analysis. For a dynamic derivation, one would prepare use cases that require a certain component, instrument the source or object code, execute the program, and then use a profiler to extract the executed operations of a component as dynamic traces for each program run. The advantage of dynamic analysis is that it yields precisely what has been executed. The problem of aliasing, where one does not exactly know at compile time what gets indirectly accessed via an alias, does not occur for dynamic analysis. Moreover, infeasible paths, i.e., program paths for which a static analysis cannot decide that they can never be taken, are excluded by dynamic analysis too. On the other hand, dynamic analysis lacks from the fact that it yields results only for one given input or usage scenario. In order to find all possible dynamic traces of the component, the use cases have to cover every possible program behavior. However, full coverage is generally impossible because there may be in principle endless repetitions of operations. Static analysis on the other hand may derive all possible traces, so-called static traces, regardless of the actual input. Static traces represent the statically derived potential execution sequences of a component's operations. As operation, one can consider any usage of a resource exported by the component, including subprogram calls, access to a global variable of the component's interface, and access to a record component of any type provided by the component. The connection between static and dynamic traces is that each dynamic trace is an instance of a static trace. To put it differently, static traces can be viewed as a grammar and a dynamic trace is a word derived from this grammar. However, in many cases, static analysis can only safely extract static traces by making conservative assumptions on the program because many questions relevant to traces, like aliasing and infeasible paths, are generally undecidable at compile time. In the view of static traces as a grammar, one can observe that static traces may often be considered a grammar that defines a superset of the actually possible dynamic traces. A static trace that was extracted via infeasible paths or an overestimation of aliasing and that cannot really occur at runtime will be called an infeasible static trace in the following. Both static and dynamic analyses are, thus, approximations where static analysis yields the upper bound of all possible traces and dynamic analysis the lower bound. Some of the underlying techniques and information needs were presented in section 2.1 of this report. These traces serve as the foundation for most of the dynamic analysis techniques introduced in this section and for some of the static approaches.

From a component substitution view point, three major dependencies become a factor to establish the interconnection and dependencies among different components [KOT05].

Intra-component information can be applied to identify dependencies between operations of a component. These dependencies may help to decide whether individual traces of instances may be considered independent. Side-effect and ripple effect analysis can be applied to identify additional dependencies. These techniques require that source code is available and are discussed in more detail in section 2.3.2

Extra-component information is based on the static traces and the information and dependencies that can be identified as part of the collection of static traces.

Domain knowledge is needed to relate the operations to the application domain and to understand their semantics. Protocol recovery is semi-automatic; domain knowledge is integrated by way of the user who recovers the protocol.

A protocol recovery process can be seen as an iterative, interactive process that uses extracted static traces as a starting point with the goal to unify them into protocols. In this interactive process, a user triggers an automatic analysis that identifies (potential) opportunities where static traces can be unified and validates them. Since both static traces and protocols are represented as finite state automata, the unification is a set of graph transformation

rules, where the semantics of these transformations can be specified in terms of the underlying language theory for finite state automata. Two types of alternative transformations can be identified:

- semantically preserving transformations, i.e., transformations that do not change the language of the finite state automata,
- and transformations that do change the language of the finite state automata but that could still be allowable.

Semantics-preserving transformations include the conversion of non-deterministic automata into deterministic ones and minimization of the finite state automata by way of Moore's algorithm. If two static traces are not completely equivalent or subsume each other, Moore's algorithm at least identifies common suffixes. A reversed version of Moore's algorithm is also able to identify common prefixes.
In the category of transformations that do not maintain the semantics but that could still be allowable, one can consider reordering of operations if they do not have any data dependency according to intra-component data flow analysis. Note the absence of data dependency is not always sufficient to decide whether reordering does not effect the semantics of the program, as exemplified by the following example of two operations, in which *is_initialized* must be called before *is_empty* even though there is no data dependency between these functions:
Another non-semantics-preserving transformation is, for instance, marking an operation that does not change the state of a component as optional if no other operation is control-dependent on it, which may trigger further transformations. Additional examples can be found in. The user can add information on semantic equivalence of certain operations, opening new opportunities for further graph transformations.

2.3.1.2 Protocol validation

The purpose of protocol validation is to validate that each collected trace conforms to the specified protocol. Protocols are typically checked at run-time. This becomes also an important issue during component substitution, due to the fact that the protocol of the substituted component has to have an identical behavior. However, to be on the safe side, one has to validate protocols statically. Static protocol validation has to check that every static trace is either infeasible or is covered by the protocol specification. It goes without saying that protocol validation can only be done semi-automatically, since many questions related to protocol validation are generally un-decidable. However, it would still be very useful for large systems to at least identify the potential mismatches between static traces and the specified protocol and then let the user decides whether the static traces actually do not conform to the protocol. Again, both static traces and protocols describe a language. Thus, for such a validation, one basically has to show that the language described by the static trace is a subset of the language described by the protocol. Unfortunately, verifying this property is only possible for regular languages in general. Consequently, different authors have proposed to use finite state automata to specify protocols. These protocols express the sequencing constraints on a component's operations only. Constraints on the data passed to the components, for instance, are not part of sequencing constraints. For example, it cannot be expressed with finite state automata that the element that is currently being retrieved from a container component must have been added before. In order to validate a static trace against a protocol, one can simply carry out the following procedure:

1. Represent the static trace and protocol by two deterministic finite state automata, *T* and *P*, respectively,
2. Combine these two automata by adding one new starting node, *S*, plus two epsilon transitions from *S* to the starting nodes of *T* and *P*, where the accepting states of the new combined automata is the union of the accepting states of *T* and *P*,
3. Minimize the combined automata using Moore's algorithm ,
4. And check whether every state, *t* of *T* has at least one equivalent state, *p* of *P* in the minimized combined automata and, if *t* is an accepting state, *p* is also an accepting state.

In order to validate static traces against a protocol, both static traces and protocol must exist. While static traces can be extracted automatically, the protocol needs to be specified by the programmer. However, if the original programmer did not properly specify the protocol, it needs to be derived by someone who might not be familiar with the component. This section describes how extract static traces. If the component's interface consists of global variables and subprograms only, deriving the static traces is a simple traversal of the inter-procedural and intra-procedural control flow graph that collects the accesses to the global variables belonging to the component and the

calls to the subprograms provided by the component in the order in which they occur in the control flow. If the component exports types, a programmer may create an arbitrary number of values of these types by declaring instances as global or local variables or formal parameters or via dynamically created instances on the heap. In case of instances, the static traces need to be extracted for each instance individually.

For protocol recovery based on extracted static traces, one could also ignore arrays. An even more difficult problem exists for instances created on the heap. Similarly to arrays, one can combine all instances created at a certain heap allocation. The particular heap allocation then serves as an atomic instance. Additionally, one has to track the pointers referring to the instance created on the heap in order to identify operations that involve the instance. Hence, a points-to analysis is needed. Points-to analysis is also required to precisely keep track of references to a value via aliases. The combination of static traces of all instances and further operations provided by the component make up the static traces for the component as a whole. If there are no dependencies between individual traces of instances and other operations of the component, one can simply unite the static traces. If one cannot exclude dependencies, the static trace for the component as a whole is the sequence of all operations disregarding which instance is passed, which is basically an interleaving of all individual static traces and further operations.

2.3.1.3 Compositional Component Adaptation [MCK04]

At the core of most approaches that actually perform/support component substitution process and compositional adaptation, is the notion of introducing a level of indirection that enables interception and redirection of interactions among program entities. The following description of compositional adaptation is based on the article by [A Taxonomy of Compositional Adaptation]. The realization of self-configuring systems has been aided by the confluence of three key technologies: separation of concerns, computational reflection, and component-based design (Figure 7). Together, they provide programmers with the tools to construct self-adaptive systems in a systematic and principled (as opposed to ad hoc) manner. In addition, the widespread use of middleware in distributed computing has served as a catalyst for research in compositional adaptation, since middleware is a natural place to locate many types of adaptive behavior.

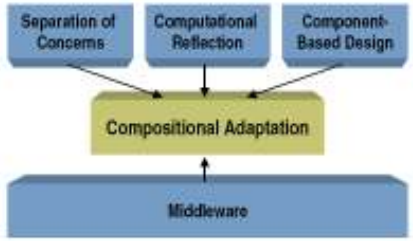


Figure 7 Compositional Component Adaptation

Separation of concerns enables the separate development of the functional behavior and the crosscutting concerns (e.g., quality of service, fault tolerance, security) of an application. The functional behavior of an application is sometimes referred to as its *business logic*. This separation simplifies development and maintenance, while promoting software reuse. Separation of concerns has become an important principle in software engineering, and many development techniques apply it to some degree. Examples include domain-specific languages, generative programming, generic programming, constraint languages, feature-oriented development, and aspect-oriented programming. Presently, the most widely used approach appears to be aspect-oriented programming (AOP). Kiczales et al. recognized that complex programs include various *crosscutting concerns* (properties or areas of interest such as Quality of Service (QoS), energy consumption, fault tolerance, and security). While object-oriented programming abstracts out commonalities among classes in an inheritance tree, crosscutting concerns are scattered among different classes, complicating the development and maintenance of applications. AOP enables separation of crosscutting concerns during development of the software. Specifically, the code implementing crosscutting

concerns of the system, called *aspects*, are developed separately from other parts of the system. In AOP, locations in the program where aspect code can be woven, called *pointcuts* are typically identified during development. Later, for example, during compilation, an *aspect weaver* can be used to weave different aspects of the program together to form a program with new behavior. AOP proponents argue that disentangling crosscutting concerns leads to simpler development, maintenance, and evolution of software. Examples of AOP approaches include AspectJ, Hyper/J, DemeterJ and Composition Filters.

Computational reflections refers to the ability of a program to reason about, and possibly alter, its own behavior. From a component substitution perspective, reflection enables a system to reveal (selected) details of its implementation without compromising portability. In other words, reflection exposes a system implementation at a level of abstraction that hides unnecessary details, but still enables changes to the system behavior. Reflection comprises two activities. *Introspection* enables an application to observe its own behavior and *intercession* enables a system or application to act on these observations and modify its own behavior. In a self-auditing distributed application for example, introspection would enable software "sensors" to observe and report usage patterns for various components. Intercession would enable new types of sensors, as well as components that implement corrective action, to be inserted at run-time.

As depicted in Figure 8, a reflective system (represented as *base-level* objects) and its self representation (represented as *meta-level* objects) are *causally connected*, meaning that modifications to either one will be reflected in the other. A *meta-object protocol (MOP)* is an interface that enables "systematic" (as opposed to ad hoc) introspection and intercession of the base-level objects. Moreover, MOPs can be categorized as enabling either structural or behavioral reflection. *Structural reflection* addresses issues related to class hierarchy, object interconnection, and data types. As an example, an object can be examined to determine what methods are available for invocation. Conversely, *behavioral reflection* focuses on the computational semantics of the application. For instance, a distributed application can use behavioral reflection to select and load a communication protocol well-suited to current network conditions.

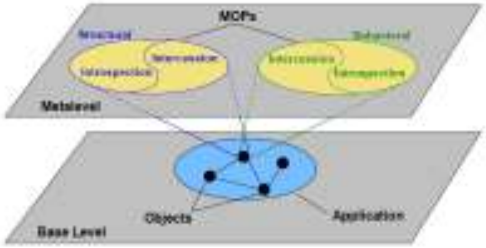


Figure 8: Metalevel understanding collected into metaobject protocols (MOPs) [McK04]

An adaptive system uses reflection to observe and "reason" about changes to its structure and behavior. Support for reflection may be provided by a programming language or may be implemented directly in a middleware platform. It has to be noted that reflective programming languages are concerned with the representation of a programming language (*e.g.,* constructs, implementation, and interpretation of a programming language), while reflective middleware approaches are concerned with the representation of middleware services (*e.g.,* host-infrastructure, distribution).

Much of the recent research in adaptive software focuses on adaptive *middleware*. Middleware is commonly defined as a layer of services separating applications from operating systems and network protocols. Schmidt [3] decomposed middleware into four layers, depicted in the Figure below. *Host-infrastructure middleware* resides atop the operating system and provides a high-level API that hides the heterogeneity of hardware devices, operating systems, and to some extent network protocols. *Distribution middleware* provides a high-level programming abstraction, such as remote objects, enabling developers to write distributed applications in a way similar to stand-alone programs. CORBA, DCOM, and Java RMI all fit in this layer (Figure 9). *Common middleware services*

include fault tolerance, security, persistence, and transactions involving entities such as remote objects. Finally, *domain-specific middleware* is tailored to match a particular class of applications [PIZ04].

Figure 9: Schmidt's middleware layers.

Since the traditional role of middleware is to hide resource distribution and platform heterogeneity from the business logic of applications, it is a logical place to put adaptive behavior related to various crosscutting concerns such as quality-of-service, energy management, fault tolerance, and security policy. Most adaptive middleware approaches are based on an object-oriented programming paradigm and derive from popular object-oriented middleware platforms such as CORBA, Java RMI, and DCOM/.NET. Many techniques work by intercepting and modifying messages.

Table 11 Middleware patterns

Technique	Description	Examples
Function pointers	Application execution path is dynamically redirected through modification of function pointers.	Vtables in COM, delegates and events in .NET, callback functions in CORBA, ACE
Wrapper pattern	Objects are subclassed or encapsulated by other objects (wrappers), enabling the wrapper to control method execution.	PCL, Adaptive Java, ARCAD, TRAPJ, R-Java, QuO, ACE, MetaSockets
Proxy pattern	Surrogates (proxies) are used in place of objects, enabling the surrogate to redirect method calls to different object implementations.	OGS, FTS, IRL, TAO-LB, ACT, Squirrel/InfoPipes, AspectIX, ACE, Rocks, Gnunnii
Strategy pattern	Each algorithm implementation is encapsulated, enabling transparent replacement of one implementation with another.	PCL, TAO, ZEN, CIAO, DynamicTAO, UIC, ACE
Virtual component pattern	Component placeholders (virtual components) are inserted into the object graph and replaced as needed during program execution.	TAO, ZEN, CIAO
Meta-Object Protocol	Mechanisms supporting intercession and introspection enable modification of program behavior.	Open Java, FRIENDS, PCL, Adaptive Java, AspectJ, Composition Filters, ARCAD, TRAPJ, JOIE, Kava, R-Java, DynamicTAO, UIC, Open ORB/COM, FlexNET, OpenCORBA, MetaSockets, PHOSE, Ignunn/J, Gnunnii
Aspect weaving	Code fragments (aspects) that implement a crosscutting concern are woven into an application dynamically.	AspectJ, Composition Filters, ARCAD, TRAPJ, AspectIX, PHOSE
Middleware interception	Method calls and responses passing through a middleware layer are intercepted and redirected.	FTS, IRL, TAO-LB, ACT, Eternal, Rocks, Racks, PHOSE, Ignunn/J, Gnunnii
Integrated middleware	An application makes explicit calls to adaptive services provided by a middleware layer.	OGS, QuO, TAO, ZEN, CIAO, DynamicTAO, UIC, Open ORB/COM, FlexiNET, Squirrel, AspectIX, OpenCorba, Electra, Orbix/Isis, Orbix/E, ACE, Ensemble, MetaSockets, Personal Java, Embedded Java

2.3.1.4 Wrappers and Glue Code

The majority of existing organizations need to address the issue of maintaining their legacy systems. Only very new organizations (e.g., start-up companies) succeed in developing from scratch both the backend information system and Web applications. Conversely, banks, public administrations, manufacturing companies, etc., already have their own information systems, very often with all the relevant data about customers, products and services. What they need is integration and substitution, in a seamless way, of data and applications already running on their legacy systems. Legacy systems are defined as applications of business critical value that have been in production. Therefore, most applications currently in production can be considered as legacy. Among the different reengineering strategies that have been proposed for dealing with legacy applications, the following two can be considered in the development of applications:

- Integration: consolidate the legacies into the current and future applications.
- Gradual migration: re-architect and transition the legacy system gradually.

The integration approach allows accessing legacy data and attempts to integrate legacy and new applications requiring only minimal modifications to the legacy system. The result of integration and substitution is a final composite system where the old applications are not replaced. The migration approach, conversely, produces a new system that completely replaces the old one, possibly by using intermediate and partial integration steps. During the substitution, a gradual migration is a viable option if the time constraints are not very strict and only inside single organizations. Inter-organizations architectures can only be based on integration, since it requires only the specification of interfaces and respects the autonomy of individual organizations.

Enterprise application integration (EAI) approaches are suites of tools and technologies with which to build interoperation solutions. These products provide standard data and transaction integration layers that are placed on top of database and legacy systems. But these solutions need to be built with low-level tools or tailored from their libraries of translators between common DBMSs and applications (e.g., ERPs). EAI technologies are commonly used to build access wrappers (they often support their (semi-)automatic generation), whereas the development of integration wrappers requires adhoc artifacts. Another type of legacy information system emerging in recent times are Web sites, and a vast amount of research is carried out about the problem of developing tools for data extraction, integration, and transformation from Web sources. Such tools, which are also commonly referred to as wrappers, are intended to produce structured data (e.g., database tables or XML documents) starting from HTML pages of a data-intensive site. Until now, the manual wrapper approach is the most largely used, whereas the current research efforts are towards the development of automatic wrapper generators, that is, systems which, starting from HTML pages of a particular site and some a priori knowledge about page organization, generate the specific wrapper (for that site), based on inference on labeled examples provided to the system.

A widely accepted approach to integrate and substitute COTS components within a given system is by using wrappers to provide an additional layer of abstraction with a well defined and specified interface. A wrapper is piece of code that the integrator builds to isolate the underlying COTS component from other components of the system.

Software wrapping requires an additional layer of software, called a wrapper, to encapsulate the COTS component. Wrapper s are responsible for intercepting any input that the system might send to the component or output that the component would send back to the system. In this context, one can define input and output as information being given to and returned from the component at the public interface level. The hope is to isolate the COTS component during the testing process in order to understand whether or not it is interacting with the rest of the system as expected. Once access to the component's input and output is gained, a variety of testing operations can be performed to provide an improved understanding of the component's interaction with the rest of the system. For example, the wrapper can be hooked up to a mechanism that simply monitors, records, and stores the input and output of the component for observation. This can be particularly helpful when the system is comprised of multiple COTS components. Consider the situation where one COTS component's output becomes another COTS component's input and source code is not available for either component. If both components are encased in wrappers, then the person testing the system gets a glimpse as to what is occurring when those two components interact. This ability to observe the interactions among different COTS might be specifically beneficial during component substitution at the feature/package level.

Component wrapping and glue code are some of the basic approaches to integrate, decouple and substitute components. There are two major types of wrappers that can be distinguished:

- **Access wrappers** provide a view of existing access functionality, by providing a new interface that corresponds exactly to the available data and application access paths.
- **Integration wrappers** provide a higher level of abstraction, by implementing new interfaces that do not necessarily map exactly to existing information access paths. While the external view of the information asset appears as a self-consistent schema (e.g., an object schema), its implementation relies on the coordinated use of multiple access wrappers in order to present an integrated view of the underlying data and application services, which conforms to that schema.

There are a number of reasons why system architectures should include wrappers around components [VIG97]:

- Conform to standards, e.g., CORBA wrappers around legacy systems.
- Reduce the impact to the system of changes to the wrapped component.
- Provide a standard interface to a range of components. For mature domains a wrapper may be provided by the component supplier, e.g., ODBC. Standard interfaces are a prerequisite for substituting COTS components from different vendors.
- Add (or hide) functionality of a component.
- Give system integrator control over the "look and feel" of the component. Even though the integrator has no control over the component, the integrator does have access to the source and control over the wrapper.
- Provide a single point of access to the component. One example of wrappers is the ODBC interface to the Microsoft Access database. ODBC provides an industry standard API for accessing relational databases. By restricting access to the database to be ODBC compliant, one can configure a system with a different database product by maintaining a similar data schema.

Another example of a wrapper is the script around a product development application. This wrapper provides a standard for moving data into the application. Unlike ODBC which is an industry standard, this interface will be controlled by the designer. It will allow systems to be configured with different product development applications and for new applications to be plugged in as they become available.

There exist several variations of the wrapping concept, like one can devise a method to apply wrapping to object-oriented components whose source code may not be available. In [KOS99], a component was defined as a *Java class*. The analysis to devise a solution for wrapping object-oriented components in this approach is based on the following general requirements for a wrapping prototype in mind (Figure 10):

1. Wrapping a component must not change the core functionality of any program that utilizes the wrapped component. While a wrapping approach might introduce a slight execution overhead, it should not destroy the functionality of systems that previously executed in a correct fashion.
2. The user of a wrapping tool should be able to turn the wrapping mechanism on and off at system start-up time. For example, a system integrator should be able to add a “wrapper flag” to the command line statement that starts an application in order to indicate that wrappers should become active.
4. The wrapper must be able to handle any operation that is advertised by a Java interface and provide the ability to intercept information at method entries and method exits (whether it is an expected or unexpected exit), thus allowing input and output data to be intercepted.

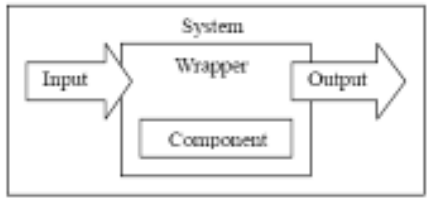


Figure 10 Component Wrappers

Glue Code [KAZ99]

One of the most significant changes in the software development area is the trend of building systems incorporating pre-existing software, with special emphasis upon the use of commercial-off-the-shelf (COTS) software components. As mentioned earlier, the challenge is that one does not have any control over the functionality, performance, and evolution of COTS products due to their black-box nature. Besides, most COTS products are not designed to interoperate with each other and most COTS vendors do not support glue code (sometimes called glueware and binding code). So, most software development teams that use COTS components have difficulties in estimating effort and schedule for COTS glue code development and integration of them into application systems. Without the glue code, the components would be un-integratable and COTS-based systems can be difficult to comprehend, less evolvable than intended, and less reliable than the original components (Figure 11).

The glue code provides the functionality to combine the different components. Purposes of the glue include:

- Control flow. Invokes functionality of the underlying components as required.
- Component bridge. Glue code can resolve any interface incompatibilities between components for example, by performing any necessary data conversions.
- Exception handling. By trapping exceptions, glue code can provide a consistent exception handling mechanism.

Within for example a distributed management system, the server side glue code might have been developed in Perl. The client it is being developed in Visual Basic. In both cases, they serve primarily as middleware combining components to add system functionality.
Glue code is considered as one of following:

1) any code required to facilitate information or data exchange between the COTS component and the application,
2) any code needed to "hook" the COTS component into the application, even though it may not necessarily facilitate data exchange, and
3) any code needed to provide functionality that was originally intended to be provided by the COTS component, and which must interact with that COTS component.

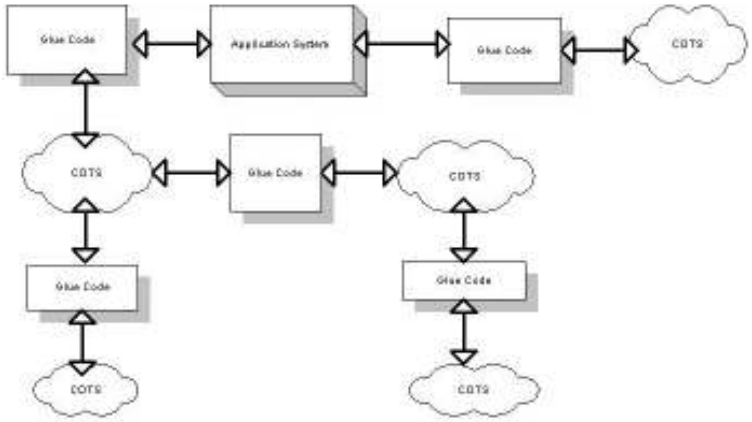


Figure 11 Dynamic Model for COTS glue code

2.3.1.5 Component tailoring [VIG96]

Component tailoring, even if not directly introduced to support component substitution, can provide some general guidelines to support component substitution. Component tailoring refers to the ability for system integrators to enhance the functionality of a component in ways that are supported by the component vendor. The tailoring is done by adding some element to the component to provide it with functionalities not provided by the vendor. Tailoring does not involve modifying source code of the component. Examples of tailoring include "scripting", where an application can be enhanced by executing a script upon the occurrence of some event. Early versions of scripting include simple macro languages. The scripting capability in many newer applications has become more sophisticated with full-fledged programming languages and interpreters, such as VBA, TCL and Perl, being accessible within applications. Another example of a tailoring capability is the use of plug-ins. A plug-in is a component that registers with the enclosing application. The enclosing application makes a call-back to the plug-in when its functionality is required. By publishing the registration and call-back techniques, COTS vendors provide COTS users with a method of enhancing the component functionality without access to the source code. When tailoring components in this way, designers must remember that the tailoring aspects are components in their own right. The designer must treat them as separate configuration items, make sure they are installed with the corresponding container component, and make sure that they are carried along during upgrades. One example of the use of plug-ins is for web browsers. Browsers can display only a limited number of image types typically GIF and JPEG, and do not provide editing capability for these images. By using plug-ins for Netscape or Microsoft browsers one can enhance their functionality to display more types of image formats and allow users to markup and annotate the images from within the browsers.

2.3.1.6 GUI Ripping [MEM03]

Features trees can be used to relate features with each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features. From a GUI perspective, a typical user input/output provided by a GUI can be seen as a feature provided by a system.

Memon et al [MEM03] present an approach for GUI ripping which is introduced in detail. Graphical user interfaces (GUIs) are important parts of today's software and their correct execution is required to ensure the correctness of the overall software. These GUIs may also provide insights on the interconnection and dependencies that exist among different parts of the system. In their paper, Memon et al focused mainly on detecting defects in GUIs by executing test cases and checking the execution results. Test cases may either be created manually or generated automatically from a model of the GUI. GUI testing requires that test cases (sequences of GUI *events* that exercise GUI *widgets*) be generated and executed on the GUI. However, currently available techniques for obtaining GUI test cases are resource intensive, requiring significant human intervention. The most popular technique to test GUIs is by using *capture/replay tools*, where a human tester interacts with the *application under test* (AUT). The capture component of the tool stores this interaction in a file that can be replayed after using the replay component of the tool. GUI ripping has numerous other applications such as reverse engineering of COTS GUI products to test them within the context of their use, porting and controlling legacy applications to new platforms, and developing model checking tools for GUIs (Figure 12).

Figure 12 GUI Forest (Tree) for MS WordPad.

GUI ripping is a dynamic process that is applied to an executing software's GUI. Starting from the software's first window (or set of windows), the GUI is "traversed" by opening all child windows. All the window's *widgets* (building blocks of the GUI, e.g., buttons, text-boxes), their *properties* (e.g., background-color, font), and *values* (e.g., red, Times New Roman, 18pt) are extracted. Developing this process has several challenges:

First, the source code of the software may not always be available; therefore one has to develop techniques to extract information from the executable files.

- Second, there are no GUI standards across different platforms and implementations. One needs to extract all the information via low-level implementation-dependent system calls, which are not always well-documented.
- Third, some implementations may provide less information than necessary to perform automated testing, therefore one has to rely on heuristics and human intervention to determine missing parts.
- Finally, the presence of *infeasible paths* in partial event flow GUIs prevents full automation.

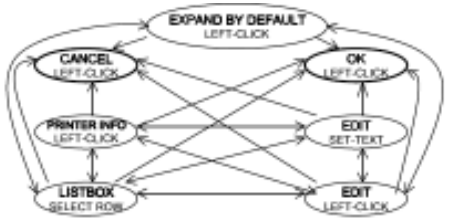


Figure 13 GUI Interaction

For example, some windows may be available only after a valid password has been provided. Since the GUI Ripper may not have access to the password, it may not be able to extract information from such windows.. One can represent the GUI's structure as a *GUI forest* and its execution behavior as *event-flow graphs*, and an *integration tree*. Each node of the GUI forest represents a window and encapsulates all the widgets, properties and values in that window (Figure 13). There is an edge from node `x` to node `y` if the window represented by `y` is opened by performing an event in the window represented by node `x`, e.g., by clicking on a button. Intuitively, event-flow graphs and the integration tree show the *flow of events* in the GUI.

2.3.1.7 Component Metrics

An inherent difficulty in performing component substitution is the need to analyze and predict the effort involved in performing a particular substitution. Metrics can provide some general measure of the maintainability, inter-operatibility and comprehensibility of components. Whether static or dynamic, there exists a lack of an agreed upon set of ideal metrics for component-based software systems. If the reliability of an individual component were low, the entire system's reliability would become also low. Therefore, it is important to measure the quality of components. There are many measurement methods (product metrics) for object-oriented software, e.g. C&K metrics [CHI91]. However, these conventional metrics require the analysis of source codes, and cannot be applied to components whose source code is not available. The following information is for example available for all JavaBeans components.

- **Attribute**: member field of component's facade class
- **Property**: attribute whose value can be observed (readable property), or attribute whose value can be updated (writable property) from the outside of each component
- **Read/Write method**: operation to observe/update the attribute's value from the outside of each component
- **Business method**: executable operation from the outside of each component. JavaBeans provides the naming conventions that enable components to be manipulated in a uniform way for the properties.

There exists several metrics which are typically applied to measure the reusability of components. These same metrics can be re-applied to provide some guidance with respect to the potential difficulties that might be involved in a component substitution. In what follows, five component reuse metrics that can also be applied towards the component substitution are briefly introduced.

One can assume that four characteristics are strongly related to the reusability of JavaBeans components: understandability, testability, usability and portability. The following is a list of five metrics, which can measure the quality of these characteristics: *EMI*, *RCO*, *RCC*, *SCCr* and *SCCp*.

(1) Existence of Meta Information

EMI ::= 1 (Bean Info class exists), 0 (otherwise) If the value of *EMI* is 1, users of components can easily understand components' usage that the components' developers assume. The value of *EMI* should be 1.

(2) Rate of Component Observability

RCO ::= Pr / A {*Pr* : number of readable properties, *A* : number of attributes}

If the value of *RCO* is in the effective interval [0.27, 0.50], understandability, testability and usability of the target component are all appropriately high. Component observability is a very important factor that affects the component testability.

(3) Rate of Component Customizability

RCC ::= Pw / A {*Pw* : number of writable properties} If the value of *RCC* is in the effective interval [0.24, 0.42], usability of the target component is appropriately high. Some attributes of the component should be able to be updated with their write methods from the outside for component customization.

(4) Self-Completeness of Component's return value

SCCr ::= Mv / M {*Mv* : number of business methods without return value, *M* : number of business methods} If the value of *SCCr* is in the effective interval [0.64, 0.96], portability of the target component is appropriately high. When all functions that the component provides are self-contained inside of the component, the independence of the component is high. One thinks business methods without return value/parameters provide such self-contained functions.

(5) Self-Completeness of Component's parameter

SCCp ::= Mp / M {*Mp* : number of business methods without parameters} If the value of *SCCp* is in the effective interval [0.33,0.68], portability of the target component is appropriately high.

2.3.1.8 Discussion – COTS component substitution at the feature/package level

Information needs:
The general information needs of the presented techniques to support COTS component substitution at the feature/package level, can be summarized as follows:

- Based on traces and the information and dependencies that can be identified as part of the collection of static and dynamic traces, as well as domain experts for protocol recovery
- Existing protocols and interfaces for protocol verification
- Feature extraction based on usage scenarios and an existing GUI (GUI ripping)
- Dependency graphs based on static/dynamic traces
- Component interfaces (including parameters, type of middle ware used, etc)

Techniques:
The techniques review in this section can be grouped in two major categories.

- Extraction techniques that provide dependency information based on existing traces (metrics, component tailoring) or recovered protocols, interfaces based on the information stored in the static or dynamic traces. Among these approaches, there is also GUI ripping that attempts to identify features based on usage scenarios and their recorded traces.
- Component wrapping, gluing, tailoring and compositional adaptation and middleware, which all have in common that they provide, introduce or utilize an additional layer of abstraction to allow for a component to be modified, changed, and substituted without affecting the remaining system.

Information provided and applicability to component substitution:
Information provided by these techniques can be summarized as identified features and usage scenarios that can be used to identify and comprehend the inter-operatibility among features in the system and therefore, provide some guidance with respect to the impact of a component substitution on the system. The metrics provide a general guideline with respect to the complexity, tightness and ease of potential modifications of the system.
The wrapper, gluing, tailoring and middleware approaches provide an additional layer of abstraction, with often well defined interfaces. These techniques are some of the traditional and successful approaches to component substitution within COTS based system, where the source code is not available. The challenge is to identify the interfaces and protocols, as well as the verification of the protocols and component features after the substitution was performed.

Table 12 Applicability of COTS analysis at the feature/package level

Technique	Application
Protocol recovery	Comprehension, impact analysis
Protocol validation	Validating/verification that component protocols match before/after substitution
Wrappers/glue code	Adaptation of legacy system, hooks, minimize change impact facilitate information and data exchange
Component tailoring	Component enhancement
GUI Ripping	Reverse engineering of COTS, adapting legacy applications to new systems, Feature extraction
Component metrics	Prediction of maintainability and substitution
Compositional component adaptation	Maintainability, comprehension

2.3.2 FOSS

There are many existing techniques for atomic component detection and comprehension in the literature. Form a component substitution view point, identifying and comprehending component at this level not only require additional abstraction of the low level detailed information provided by the tools and techniques discussed in section 2.1.2 and 2.2.2. It also requires to provide maintainers and programmers with a mental representation of these systems that closely matches either their current understanding of the system or enriches their understanding to gain additional insights. The techniques and their information needs that are surveyed in this section range from identifying components (e.g. component analysis), to analyzing the impact of changes to the existing system (e.g. impact analysis) to traceability analysis. It has to be noted, that there is no single technique that can support a general component substitution process. A combination of these different techniques anis required to provide the insights and information needed to guide maintainers during the comprehension and substitution process.

Table 13 Overview FOSS analysis at the feature/class/package level

Technique	Information need	Information provided	Application	Static	Dynamic
Change impact analysis, ripple effect, traceability, dependency analysis	Static and dynamic traces, program model, domain knowledge, system model	Ripple effect analysis, trade-off analysis, validating substitution	Comprehension, impact analysis, maintenance, protocol verification	Y	Y
Concept analysis, grouping techniques, feature extraction	Program model, domain knowledge, source code, program traces	Features, logical grouping and clustering	Comprehension, feature extraction, substitution, testing, refactoring, feature analysis	Y	Y
Behavioral analysis	Tracing, domain knowledge	Dependency analysis, communication dependency	Comprehension, component interaction analysis	N	Y
Aspect oriented	Source code, binary , program model	Separation of concerns	Refactoring, tracing	Y	Y
Data reverse Engineering	Schema, database, domain knowledge	Grouping , feature extraction, associations, dependency analysis	Program comprehension, maintenance	Y	Y
Software Visualization	Source code, traces, program model, domain knowledge	Abstraction, views	Structural comprehension, Interaction	Y	Y

2.3.2.1 Impact analysis

Once the software is understood, the proposed change must be analyzed. Since it is unlikely that an entire software will change at once, change impact analysis in some form is necessary to determine what other parts of the software may be affected if a change is implemented. Ideally, impact analysis is an iterative process that is performed as early as possible in the change cycle in order to determine the scope, cost, and risk of the change [QUE94] Change impact analysis includes both determining impacts within the software (*dependency analysis*) and between different software products used during the development process (*traceability analysis*).

An *impact* is a part of the software system determined to be affected by the change, thus requiring further inspection [BOH96]. *Impact analysis* then is "the activity of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [BOH96].

Change impact analysis is a process that may involve several steps before coming to a conclusion. The IEEE Standard Glossary of Software Engineering Terminology [IEE90] does not provide a definition for impact analysis, but Standard 1219-1998: IEEE Standard for Software Maintenance [IEE98] provides a guideline for what software change impact analysis should entail during the analysis phase of software maintenance.

The standard states that impact analysis should:

- Identify potential ripple effects;
- Allow trade-offs between suggested software change approaches to be considered;
- Make use of documentation abstracted from the source code; and
- Consider the history of prior changes, both successful and unsuccessful.

These guidelines can serve as a basis for determining what should be accomplished during the impact analysis process, but they do not necessarily define a process. Bohner and Arnold in [BOH96] and [ARN93] attempt to define the sequential steps of an ideal impact analysis process. Figure 14 provides a visual representation of a generic impact analysis process discussed in [ARN93], showing the inputs and outputs of the impact analysis approach. The process begins with a change proposal that is analyzed to plan its implementation (Figure 14). After this, the change is scoped out by determining what parts of the system are initially affected by the change implementation chosen. Once the initial impacts of the change are determined, the potential ripple effects may be determined. These ripple effects may require further investigation while implementing the change. The accumulation of the impacts and their ripple effects provide the potential impact set of the change. This impact set can then be used to plan, predict, and accomplish that change task.

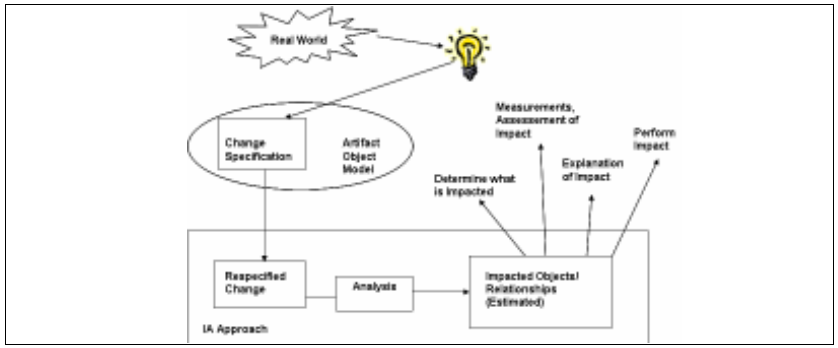


Figure 14: Generic Impact Analysis Process

2.3.2.2 Traceability Analysis

Traceability analysis is one of the two major areas of change impact analysis. Requirements traceability focuses on determining if the requirement has been implemented and if so, where? Once the requirement has been traced to the desired level of abstraction, a more specific impact analysis technique can be used to determine how the change will affect the system [BOH96]. It is said that requirements should be traceable to different design products (*vertical traceability*), within these design products (*horizontal traceability*), and also to the code that implements them [STR96]. Some say requirements traceability provides critical support for managing change in an evolving software system [SET04].

Software life cycle objects (SLOs) generated or modified during the development process are traceable if they provide the ability to associate the information between them [BOH96]. This is important because each software life cycle object, while essentially detailing the same system, provides a different view of the system at varying levels of abstraction. Thus, *traceability analysis* involves "examining dependency relationships among all types of SLOs" [BOH96]. Traceability information relies heavily on software documentation [BOH96], including but not limited to requirements specifications, designs, and user documentation [LAM98b]. The relationships between the various documentations can be illustrated in a graph structure. An example of such a graph is shown in Figure 15 [LAM98b] where the relationships between different artifacts (rectangles) within the same document (CSS Requirements Document) and then between different documents are labeled with specific identifiers (d*i*). This benefits impact analysis by providing information on what other life cycle products are affected by the proposed changes, allowing effective navigation for more detailed impact analysis [LAM98b].

There exists a body of research [STR96, SET04, and KNE03] on approaches to effectively perform traceability analysis for change impact analysis. The mentioned references focus on how to trace requirements to design documents and models.

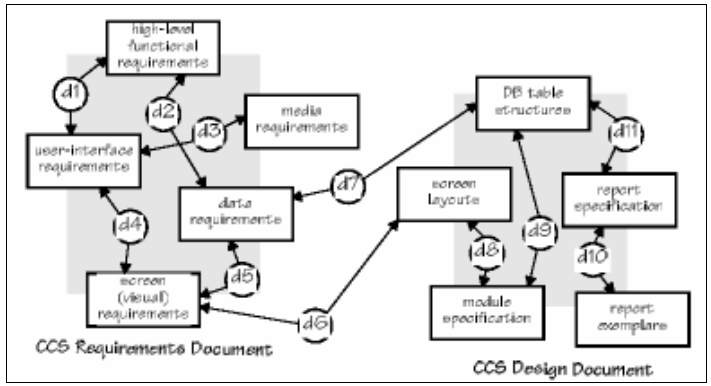


Figure 15. The Documentation Architecture

2.3.2.3 Dependency Analysis

Dependency analysis represents the second major branch of impact analysis and is used for determining the impact of changes within the system. *Dependency analysis* involves examining detailed dependency relationships among program entities. This type of analysis is narrowly focused, providing detailed evaluations at the code level, but not at higher levels of abstraction. At the time of their assessment, this may have been true, but over the years, dependency analysis approaches have been developed for higher levels of system abstraction. The following is a list of different types of dependencies between two modules A and B:

- *Syntax Dependencies:* Can be either data (type/format of data is consistent) or service (signatures of services are consistent) related
- *Semantics Dependencies:* Can be either data or service related
- *Sequence-of-use dependencies:* Can be either data or control related
- *Interface identity dependencies:* Interfaces between modules must be consistent (name)
- *Runtime location dependencies:* Must be consistent (same of different processor or located within same process)
- *Quality-of-service or quality of data dependencies:* Involves the service or data provided by the modules
- *Existence-of-module dependencies:* Either must be present for the other to function properly
- *Resource behavior dependencies:* Relates to such issues as memory usage, resource ownership between the modules

A *dependency* can be defined as all relations, D, between some number of entities wherein a change to one of the entities implies a potential change to others. Furthermore, one can also differentiate between directed dependencies and categorize them into unidirectional and bidirectional: a *unidirectional* dependency d(A,B) exists if node A depends on node B, while a *bidirectional* dependency exists if A depends on B and B depends on A (as in a recursive call). A set of attributes which are common to all dependencies including, sensitivity, stability, importance, and impact. Impact meaning in what ways is the dependent entity's functionality compromised by failure of this dependency. Dependency analysis can be performed manually or automatically using techniques like program slicing, control- and data-flow analysis, test-coverage analysis, and cross referencing to evaluate the data, control, and component dependencies between system entities. Additionally, dependency analysis may be performed on static system information, such as a class diagram or on dynamic information, such as actual execution traces of the system.

2.3.2.4 Scenario Dependency Analysis

Scenarios, defined in [RADZI] and [BRE00], have been used to describe the execution sequences of system functionality for a long time because they provide a good compromise between informal use cases and formal designs [AMY01c]. The use of scenarios allows advantages such as: validation of requirements and the comparison of requirement alternatives [KAZ96]; reduction of complexity in requirements understanding [WEI98]; and aiding in requirements agreement among different stakeholders [WEI98].

Scenario dependencies have been detailed in [TSA03], [PAU01], [TSA01], [BAI02], [BOR01] and [BRE00]. Each describes a slightly different set of scenario relationships and/or dependencies as they relate to their own research area.

Functional Dependency

Bai et al. in [BAI02] describe how to define scenario groups to create test scenarios. They discuss working from the requirements and decomposing each functional feature in order to define the groups that the scenarios may fit into. They conclude that functional dependency among scenarios exists if the scenarios belong to the same group that represents common system functionality. Figure 16 [BAI02] shows an example of the scenario model in which the functions of a Banking System are decomposed progressively into multiple levels of scenario groups, scenarios, and sub-scenarios.

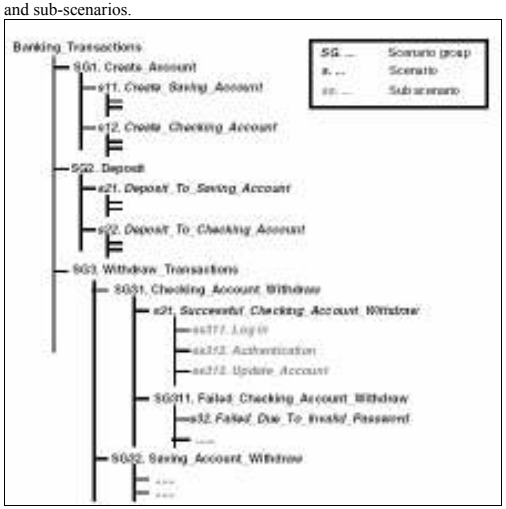


Figure 16: Example of Modeling Scenarios

A predecessor of this research is [TSA01], where discussion pertains to arranging thin threads hierarchically for the purposes of generating test cases, performing risk analysis, and accomplishing ripple effect analysis. In their approach, a thin thread tree is created and the root of the tree is the system under test. Each branch of the tree represents a group of scenarios related by functionality, while each leaf represents a concrete scenario. This, they claim, effectively creates a functional decomposition of the system.

Containment Dependency

In [PAU01] and [BAI02], a "path-contained relationship" for a scenario execution path is defined. For two scenarios A and B, if the complete path of A is part of the path of B, then A is contained in B and B depends on A. Similarly, in [BRE00] a subset relationship is discussed. They define a scenario to be a subset of another if one scenario shares the context of the other scenario.

Condition Dependency

In [BAI02], scenarios are said to be condition dependent if they are affected by the same conditions. Comparably, [PAU01] specifies that scenarios are condition dependent if they share pre- and or post-conditions. Additionally, in these works, the relationships that exist between the conditions themselves are defined. Some of the relationships defined for conditions include independent - two conditions can happen irrespective of the other, mutually exclusive – two conditions cannot exist at the same time, and related – the two conditions are used in the same thread.

Execution Dependency

Execution dependency is discussed by Paul in [PAU01] and Tsai et al. in [BAI02] within the context of generating test scenarios. They briefly define execution dependency to exist between scenarios that interact through their execution paths and share common software components such as code modules or interfaces. In [PAU01], Paul claims that scenarios may be dependent on each other in 3 ways: identical paths, covered paths, and crossing path, but provide no further definition of these dependencies.

2.3.2.5 Ripple Effect Analysis

Ripple effect analysis is a sub-process within impact analysis. A *ripple effect* is "the effect caused by making a small change to a system which affects many other parts of a system" [BOH96][ARN93]. This definition is not limited to the source code level and can be extended to include design and specification models as well [WAN96].

Figure 17 visualizes the iterative generic ripple effect analysis process defined in [WAN96] that includes: 1) Making the initial change; 2) Identify potentially affected areas due to that change; 3) Determine which of these areas needs further changes; and 4) Determine how to make that change.

Ripple effect analysis (REA) focuses on determining what parts of the system may be affected by a change. Using a chosen approach, the location of the initial change is identified and the effects of that change are recorded to create the impact set. How the impact set is identified depends on the approach itself that either uses dependency analysis or traceability analysis as its basis.

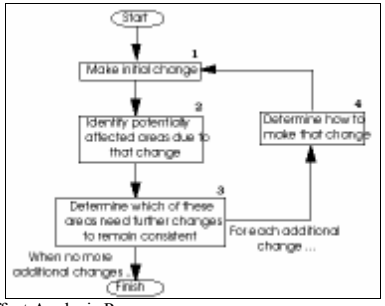


Figure 17: Generic Ripple Effect Analysis Process

To determine the impact associated with a change, the system model must be searched to identify relationships among system entities. In [BOH96] and [BOH02], the following summary of several search algorithms which form the basis of many ripple effect analysis approaches is provided. They state that some search algorithms may be semantically guided where the impacts are obtained from a predetermined semantic network of objects. Others may be heuristically guided in which a predetermined set of rules suggest possible paths that may contain impacts. Stochastically guided searches use a given situation as the basis for determining the probabilities of impact. Hybrid searches combine the aforementioned algorithms, while unguided searches attempt to find impacts in a brute force manner. Transitive closure used on call graphs is used as the basis for many of these algorithms to determine the impact sets of a change [LEE00][BRI03][LAW03].

2.3.2.6 Change Impact Analysis (IA) Approaches

A significant amount of research on change impact exist, ranging from analysis at the source code level up to the system architecture level. Queille et al. in [QUE94] identify three issues that need to be resolved when performing impact analysis: First, the system must be represented to show the system objects and the relationships between these objects. A dependency graph is an example of such a system representation. Next, the data to populate this model must be collected. Finally, a method of tracing through this representation, such as a call graph to determine the impact set, must be created.

In general it can be assumed that most impact analysis approaches use either dependency or traceability information as their basis for impact determination. In what follows major impact analysis approaches are reviewed.

2.3.2.6.1 Source Code-based Approaches

Source-code change impact analysis uses the source code of the system as its basis for determining relationships within the system. These approaches have the advantage of being very accurate in the analysis, since they identify impacts in the final product. However, they have the disadvantage of being extremely time consuming, limited in scope, and they require implementation of the change before the impact can be determined [BRI03].

Kung et al. in [KUN94] discuss the type of code changes that can occur in object-oriented classes (class changes, method changes, etc.) and provide a solution to determine their impacts. Furthermore, Lee et al. in [LEE00] define object-oriented data dependency graphs (OODDG) that emphasize data relationships relevant to object-oriented systems between such items as classes, class members, and constants. The OODDG actually consists of three graphs specific to method, inter-method, and inter-classes relationships in the system. This work provides algorithms to evaluate changes and metrics to quantitatively evaluate change impacts.

Similarly, Kim et al. in [KIM99] investigate the impact of changes to object-oriented software in distributed environments and propose a distributed program dependency graph (DPDG) that shows relationships that include data, design documents, servers, and classes. They argue that conventional change impact analysis is hard to apply to distributed systems because their characteristics are limited to centralized system environments.

Other approaches focus on determining impacts from a dynamic representation of the system, where a representation is created based on execution traces. Law and Rothernel in [LAW03] propose a novel approach for impact analysis using "whole path profiling", based on dependency analysis at the procedure level. It uses low cost instrumentation to retrieve dynamic information about system execution and then builds a representation from that collected information. This approach incorporates call-order and call-returns in the impact set. Similarly, work in [ORS03] takes this one step further and provides algorithms for impact analysis using instrumentation information from deployed software. They claim that their approach provides better results for impact sets than using fabricated system executions.

2.3.2.6.2 Slicing Based Approaches

Program slicing can be used to determine the potential effects of making a change [WAN96]. Static slicing determines dependencies for all program inputs while dynamic slicing searches the dependency graph for dependency based on the input given [LAW03]. The following discussion summarizes some of the major approaches that make use of slicing algorithms to compute impact sets.

In [TON03], a novel approach is introduced that combines slicing and concept analysis called "Concept Lattice of Decomposition Slices". The authors claim that the approach is an extension of the decomposition slice graph in that the graph is obtained from concept analysis and it can be used to assess change impacts at given program points. Similarly, in [CHE96], researchers discuss how to create a dependency graph from a C++ program (termed "C++ Program Dependence Graph") to capture declaration, message, and class dependencies relevant to object-oriented systems. Impacts on this representation are determined by the application of developed C++ specific slicing algorithms to slice classes, messages, and programs.

Some other works have applied slicing algorithms to system models instead of a dependency graph created from the source code. For example, in [KOR03], an approach is presented that applies slicing to extended finite state machines (ESFM) in effort to analyze the system with respect to a particular functionality. They create an ESFM dependency graph using the data and control dependencies of the model and suggested that this approach can help in understanding how the model will interact with changes made to the system. Similarly, in [HEI98], slicing is applied to hierarchical state machines with the goal of performing impact analysis on the Requirement State Machine Language (RSML).

2.3.2.6.3 Model-based Approaches

There is a growing trend towards model-driven development. It is creating a need to perform change impact analysis on a representation of the system that is at a higher level of abstraction than source code [SET04]. The benefit of these model-based approaches is their ability to determine impacts without implementation of the change, although they may provide less precise results [BRI03].

In [BRI03], Briand et al. propose a static, UML model-based approach to impact analysis that can be applied before changes are implemented to help in the planning process. They first check for consistency between the diagrams and the implemented system. Then, changes between the two models (the original and the changed) are identified using a "change taxonomy" to associate their formally defined (using Object Constraint Language) impact rules with types of changes. Finally, impacts are determined using the defined impact rules and transitive closure.

A dynamic impact analysis approach that supports understanding of the effects of changes on an ESFM is presented in [KOR04]. Using both the original model and the modified one, they provide an algorithm for automatically determining the differences between the two models. The algorithm identifies parts of the model that may be affected by the change using model-based dependency analysis.

Finally, Zhao et al. in [ZHA02] propose an approach for change impact analysis that is at the architectural level. Based on an architectural slicing and chopping technique, they use a formal description of the architecture based on the WRIGHT Architectural Description Language (ADL) and automate the change impact analysis using architectural slicing and architectural chopping.

2.3.2.6.4 Discussion of Change Impact Approaches

In [ARN93] a framework is proposed for comparing different impact analysis approaches and assessing their effectiveness. This framework focuses on three main areas that can be used to compare and evaluate various impact analysis approaches regardless of the abstraction level. The areas can be summarized as follows:

- *IA application* examines how an approach is used to accomplish impact analysis by assessing the inputs and outputs of the approach.
- *IA parts* is concerned with the functional parts of the impact analysis approach – the internal model used to represent the system and its dependencies, the rules and semantics of relationships between the model entities, and the algorithms for determining the impact set.
- *IA effectiveness* is concerned with how well the approach accomplishes impact analysis. The estimated impact set is compared to the actual impact set to assess its effectiveness.

These general guidelines of how impact analysis can be accomplished provide a general understanding of what the goals of an approach should be. Some of the main features of the presented approaches are shown in Table 14.

Change impact analysis can be performed at various levels of abstractions. Impact analysis approaches (at any level of abstraction) can be categorized into two main categories termed *static* and *dynamic*, as defined in [BOH96]. Static impact analysis analyzes the source code structure without executing the code. Since it is independent of any particular program execution, static analysis considers all potential system executions and all entities that are related to the changed entity are added to the impact set. The disadvantage of this method is that a large, possibly inaccurate impact set may be returned [BOH96].

Dynamic impact analysis, on the contrary, relies on an execution trace of the system [BOH96]. Dynamic analysis has the benefit of returning an impact set that is determined from tracing through an execution of the system, implying that the results returned are almost certainly impacted [LAW03]. The disadvantage is that the execution trace(s) used may not account for all the possible executions that involve the changed entity. This results in an impact set that may not contain all the impacted parts of the system [BOH96].

In general, the level of abstraction will typically result in a trade-off between accuracy and scalability/applicability of the impact analysis approaches. The more fine grained (detailed) information is available (e.g. source statement level) the more detailed and accurate the analysis is going to be. However the cost associated with such a detailed analysis can be significant and limit the applicability of these approaches.

Conversely, model-based impact analysis approaches require a representation of the system to identify change impacts. When impact analysis is performed on an abstraction of the system, the assumption is that the model is up to date and consistent with the code. Although this can be verified using traceability tools [BRI03], there is still the possibility of inaccurate representation. Furthermore, not all developed systems are modeled using a design

notation, thus change impact analysis at this level of abstraction may require the extra step of reverse engineering the system to the required model.

The following table provides a general overview of the surveyed impact analysis techniques and their basic approach to determine the impact.

Reference	IA Application	IA Parts		
	Object Domain	Internal Model	Impact Model	Impact Tracing Algorithm
[KIM99]	Distributed object-oriented source code objects	methods, data members, classes, servers, documents	Distributed Program Dependency Graph (DPDG) with data and control dependencies	semantically guided searching
[LEE00]	object-oriented source code objects	data members, classes, methods	Object-oriented system dependency graph with data and control dependencies	transitive closure on graph
[LAW03]	source code methods	executed class methods	whole path directed acyclic graph	forward and backward searching guided by execution traces
[TON03]	source code objects	statements, data members	concept lattice of decomposition slices	forward slicing on lattice nodes
[CHE96]	object-oriented source code objects	methods, classes, objects	C++ dependency graph with message, class, and declaration dependencies	forward and backward slicing
[KOR04]	EFSM system model	system states and transitions	data and control dependence graph	transitive closure on graph
[BRI03]	UML conceptual and class models	classes, interfaces, sequence messages, state machines	UML models (sequence, state or class diagrams)	transitive closure
[ZHA02]	system architecture objects	architecture components	WRIGHT architectural structure	architectural backward and forward slicing and chopping

Table14: Change Impact Approaches Comparison

2.3.3 Ripple effect analysis and Regression Testing

An important issue in software testing is regression testing, which involves repeating the execution of a test suite after software has been changed (component substitution), in an attempt to reveal defects introduced by the change. One reason this is important is that it is often very expensive. If the test suite is comprehensive, then it can take significant time and resources to conduct and evaluate the new test. Some commercial software developers, therefore, routinely do regression testing after even minor changes, and it is not unusual for each regression testing session to run overnight (if it is automated) or even longer (if human attention is required to decide whether the software is responding appropriately). Component-based software seems to offer a way to address this problem, because a common type of change in well designed component-based systems is for one component to replace another. Even in systems that are not consciously designed for plug-in compatibility, if changes are incremental and a group of changes is limited to a single component, the effect is *as if* one component had replaced another. Is it possible to confine regression testing of software that has undergone component-level or component scoped maintenance to the "vicinity" of the replaced or modified component? Or does the entire system have to be tested again?

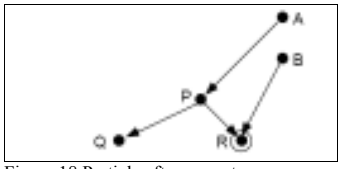


Figure 18 Partial software system

The following simple example is used to illustrate the application of regression testing on a component based system. The components include some operations with side effects on their arguments and/or global or static data. Figure 18 shows a partial view of the system. Nodes represent operations and arcs represent calling relationships.. For example, P calls Q and R; B calls R; etc.

Factors Affecting Soundness

A component can be referred to as clean if its observable behavior does not depend on any "static" data values kept privately within the component. The issue here is not whether P and P' themselves are clean or dirty, but whether the components they rely on are clean or dirty. Both P and P' call R, which is dirty. These calls cause no problem if P and P' give identical answers to their callers. Another possible question could be: What about other calls to R from elsewhere in the system, e.g., from B? Similarly, the indirect calls from P' to U (through S) might affect the outcomes of other calls to U, e.g., from C.

The "Expanding Module Phenomenon"

The above problems might be surmountable, if one expands the hypothetical "module boundary" through which calls are logged so the interior includes more than just P itself. You can record all calls to P (P') that cross the expanded module boundary, e.g., from A; all calls across the boundary to the dirty component R, e.g., from B; all calls across the boundary to the dirty component U, e.g., from C; and all calls across the boundary to the clean component S whose behavior depends on that of the dirty component U, e.g., calls from D.

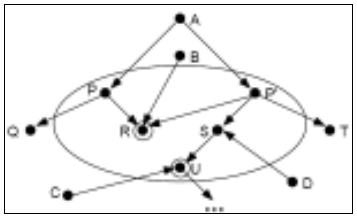


Figure 19 Expanding Module Phenomenon

The expanded module boundary is defined by constructing all call paths starting from P and P' within the component boundaries (Figure 19). This subassembly of (actual) components then becomes a (mythical, from the standpoint of the actual source code) new "component". In other words, P' cannot be regression-tested alone. It is necessary to regression test the entire expanded module induced by the proposed substitution of P' for P.

2.3.4 Clustering and Feature Extraction Approaches [JAI99, KOS99]

Clustering analysis can be described as an organization of a collection of patterns (usually represented as a vector of measurements or a point in multidimensional space) into clusters based on similarity. One of the major source of difficulty in developing, delivering, and evolving successful software is the *complexity gap* that exists between the problem and the solution domains. Evolution focused solely on the problem domain may lead to

changes that degrade the structure of the original code. Similarly, evolution based solely on technical merits could create changes unacceptable to end-users.

Clustering techniques are one major approach to support the identification of components and features at the source code level. Existing clustering approaches can be categorized based on the information they leverage. The challenge is to identify techniques that cluster the source code in a way that provide (1) meaningful logical groups that correspond to components in the source code. These approaches can be grouped in the following main categories:

- Domain-model-based approaches use a domain model that describes the domain concepts and their relationships and that is used to guide the search for components
- Dataflow-based approaches leverage dataflow information [ROU05]
- Structure-based approaches are based on base entities and their structural relationships. Structure-based approaches can further be distinguished into connection-, metric-, graph-, and concept-based approaches

Connection-based approaches cluster entities based on a specific set of direct relationships between entities to be grouped. These direct relationships allow to identify different type of entities based on their connection type (Table 15).

Table 15 Connection-based approaches [KOS99]

	Connected_Entities
Same Module	signature types, i.e., parameter or return types of a subprogram, and accessed variables declared in the same module
Part Type	signature types that are not a part-type of another type in the same signature; T is a part-type of T' if T is (transitively) used in the declaration for T'
Internal Access	signature types and variables whose record components are accessed
Same Expression	accessed variables that occur in the same expression

Metric-based approaches cluster entities based on a metric using an iterative clustering approach. Schwanke's Similarity Clustering [SCH91] is aimed at finding related subprograms based on direct call relationships among the subprograms and common and distinct usages of non-local names. Belady and Evangelisti's approach groups related subprograms using a similarity metric based on data bindings. A data binding is a potential data exchange via a global variable. Hutchens and Basili extend Belady and Evangelisti's work by using a hierarchical clustering technique to identify related subprograms and subsystems.

Delta-IC [CAN99] is actually a hybrid of connection-based and metric-based approaches since it consists of two parts: cluster formation and cluster filtering. The actual *cluster* is built around a subprogram *S* and comprises the *closely related subprograms* of *S*, i.e., all subprograms that access only variables accessed by S, and the referenced variables of *S*. The metric measures references from outside the cluster (coupling) and occurrences of minimal subclusters consisting of subprograms that reference exactly one of the variables in the cluster (cohesion).The metric-based approaches differ from connection based approaches by the degree of freedom that is offered by the metrics parameters and the threshold that can be varied to find atomic components with varying reliability.

Graph-based approaches derive clusters from a graph by means of graph-theoretic analyses. The difference to connection-based approaches is that the whole graph has to be considered whereas connection-based approaches regard only direct relationships between entities in order to decide whether they should be grouped. *Strongly Connected Components Analysis* considers cycles in the call graph as components and *Dominance Analysis* identifies local utility functions of atomic components.

Concept-based approaches use concept analysis to compute a lattice of concepts. A concept is a maximal set of objects sharing common attributes where each object has all attributes. In order to find atomic components with concept analysis, a subprogram is considered an object in the sense of concept analysis.

The following sections will review some of these clustering and feature extraction techniques in more detail.

2.3.4.1 Concept Analysis

Concept analysis is one technique which has gained momentum in the last few years as an approach that can provide logical groupings for various aspects in the software maintenance domain. Concept analysis has also applications in software comprehension, refactoring legacy systems and it can provide support during component substitution (component identification/recovery, feature extraction, refactoring). The recent success of concept analysis is based on mainly two factors. Tt is a relative lightweight analysis approach which is programming language independent and it is based on a clear mathematical foundation [TIL03, TON04]. The mathematical theory of Formal Concept Analysis was founded by Birkoff in 1940 [BIR67] and the related data analysis method was introduced by Ganter and Rudolf Wille in 1982 [WIL82]. In essence, FCA can provide support in identifying sensible groupings of objects with common attributes. There are several applications of Formal Concept Analysis (FCA). FCA has proven to be a cost-effective alternative to many applications in business and software engineering. It has been applied in various computer science branches such as conceptual clustering, data analysis, information retrieval, knowledge discovery, ontology engineering, and software engineering. This section illustrates the applicability of FCA for different application domains.

In software engineering, Formal Concept Analysis has been applied to program understanding and maintenance. Several concept-based applications use FCA as an approach to perform program module identification, which can be further developed into the application to aid in program restructuring. FCA has also been applied to re-engineering class hierarchies, module structure and software configurations. Furthermore, program feature location can be conducted by FCA. The extracted information aids programmers in program understanding, debugging, and modification. Some examples of applications of Formal Concept Analysis in software engineering are listed below.

- *Re-engineering Class Hierarchies*
  Snelting and Tip present in [SNE00] an application of FCA to class hierarchies reengineering. The goal of their research was to find imperfections in the design of class hierarchies, and to build an automatic tool to aid in the process of reengineering. In their technique, they analyzed the usage of the hierarchy by a set of applications that use it. Also, "a fine-grained analysis or the access and subtype relationships between objects, variables and class members is performed" [SNE00]. By means of FCA, the design problems, such as redundant class members or classes that should have been split, were extracted along the process. Snelting and Tip create the binary relations between class members and variables and perform FCA. Next, the commonality between class members and variables was factored out. The result of the whole process of analysis is the alternative hierarchy that is behaviorally equivalent to the initial hierarchy. However, each object in the alternative hierarchy contains only the required members. They also apply this analysis to "a space-optimizing source-to-source transformation," that removed redundant fields from objects. At that time, this analysis was considered one of the most powerful techniques to reengineer class hierarchies. However, it was expensive compared to other available techniques.

- *Re-engineering of Configurations*
  Snelting applied Formal Concept Analysis to infer configuration structures from existing source code [SNE96]. He developed the restructuring tool, NORA/RECS. This tool accepts "source code where configuration-specific code pieces are controlled by the pre-processor" [SNE96]. The algorithm then computes a concept lattice which gives users the image of the structure and properties of possible configurations. The lattice produced is a structure of fine-grained dependencies between configuration threads, and the overall quality of configuration structures. The re-engineering of configurations is done by analyzing interferences which indicated high coupling and low cohesion between configuration threads. NORA/RECS then automates decomposing the source code into modules in order to resolve these interference problems. As a result, each module only deals with a cohesive subset of the configuration space.

- *Software Component Retrieval*
  Lindig proposed a technique to efficiently retrieve reusable software components by using FCA [LIN95]. In his approach, assuming that the reusable software components from a library are indexed with a set of keywords, the user incrementally specifies a set of keywords from the requirements. The selected components and the exact set of remaining significant keywords then needs to be further refined before being presented to the user. The FCA algorithm acts as a powerful mechanism to compute a lattice that describes the relation between retrieved components and significant keywords. Results demonstrated that after the FCA algorithm was applied, users could select components quickly and precisely. This method of software component retrieval also guarantees that at least one component will be found while none of the conflicting keywords will be specified.

- *Identifying Modules*
  Sniff and Reps explain in [REP97] how to use FCA to identify modules in a program that does not designate its modules explicitly. They applied FCA to identify potential modules using both 'positive information', which is the values or types that the module depends on, and 'negative information' which is the ones that the module does not depend on. Each concept obtained from this algorithmic framework represents a potential module of a program. From the calculated concept lattice, they provide the algorithm that identifies possible alternatives to partitioning the program into modules.

- *Identifying objects from legacy code* [LIN97]
  Lindig and Snelting demonstrate how module structures are presented in the lattice, and how the lattice can be used to assess cohesion and coupling between module candidates. Certain algebraic decompositions of the lattice can lead to the automatic generation of modularization proposals. The method is applied to several legacy code written in Modula-2, FORTRAN, and COBOL. Among these programs was a 100+ KLOC aerodynamics program. Deursen et al. [DEU99], on the other hand, present a method to identify objects by semi-automatically restructuring the legacy data structures. The method involves the selection of record fields of interest, the identification of procedures dealing with such fields, and the construction of coherent groups of fields and procedures into candidate classes. They conduct object identification by using cluster and Formal Concept Analysis and then illustrate their effect on a 100 KLOC Cobol system. They compare this clustering with FCA techniques. The authors concluded that FCA solves a number of problems encountered when using cluster analysis. Furthermore, FCA is more suitable for the purpose of object identification because it finds all possible combinations since it is not solely restricted to partitioning.

- *Module restructuring* [TON01]
  Tonella applies Formal Concept Analysis to program module restructuring. This technique is based on the computation of extended concept sub-partition. It provides alternative modularizations characterized by cohesion around the internal structures that are being manipulated. FCA is used as a powerful tool to support module restructuring. Tonella evaluated the ability of Formal Concept Analysis to determine meaningful modularizations in two ways. Firstly, he uses programs without encapsulation violations to assume the absence of violations as an indicator of careful decomposition. Secondly, he implements restructuring intervention in his case studies to evaluate the feasibility of restructuring and to examine the code structure before and after the intervention.

- *Types and Formal Concept Analysis for legacy systems* [KUI00]
  Kuiper et al. have implemented ConceptRefinery which is a tool that allows a software engineer to perform system modification while maintaining a relation with both the original calculated concepts, and the legacy source code. Types for variables and program parameters which are used to conduct Formal Concept analysis are obtained from type interference in COBOL. The motivation of this tool came from the assumption that the structure in legacy systems can be obtained by FCA; FCA would perform a data analysis using the facts derived from the legacy source code together with type inference from those facts. The results showed that this technique was more precise than their previous experiment which used concept analyses without type inference.

- *Architectural element matching* [WAT99]
  Water et al. introduced an automated technique for matching the architectural elements using FCA. Their motivation for developing this technique stemmed from their wanting to support forward-engineering and design activities. They wanted to provide a method to help analysts perform architectural syntheses. They found

that FCA helped the analyst in the combination process of architectural synthesis. FCA guided the analyst to match elements from a variety of sources with the same underlying parts of the software system. A concept lattice is derived from the perspectives and descriptive information about the system domain depicted the structure of matching relations.

Tonella in [TON03] compares two impact analysis techniques, a decomposition slice graph with concept lattice of decomposition slices. The decomposition slice graph partitions the program into computations performed on different variables and displays the dependence relation among the computations. The concept lattice is created by considering a program variable as an object and the attributes are the program slice of the variable. The technique begins with localizing the basic concept lattice nodes which are directly affected by the change. Then, the lattice is traversed upward to seek further impacted nodes affected by the primary ones. Figure 20 displays an example of concept lattice of decomposition slices.

One significant difference between these two techniques is that, for the decomposition slice graph, infimum (least upper bound) and supremum (largest lower bound) are not assured to be unique for any pair of nodes while the concept lattice is. Consequently, a concept lattice of decomposition slices provides more information which is useful for program understanding, in particular, for change impact analysis. The concept lattice of decomposition slices can be applied to assess the impact of change made at given program points by determining whether the change of the computation on one variable will affect the computation on another variable. The idea of this approach is based on the assumption that any change of a particular line of code will affect its upward reachable nodes on the concept lattice.

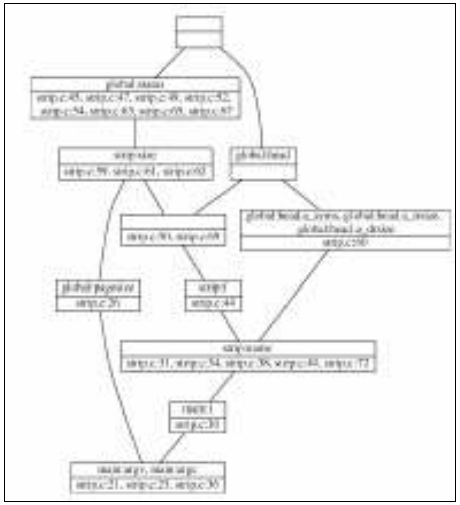


Figure 20 Concept lattice of decomposition slices [TON03]

The main advantage of the concept lattice representation over the decomposition slice graph is that the lattice presents the weak interferences among elements explicitly. The lattice representation of a program is more compact than the program itself [TON03]. However, scalability is still a challenge. Presenting a decomposition slice graph of a large program often results in information overloading which makes the graph unreadable. One possible solution is constructing a lattice incrementally and only presenting the system portion that needs to be analyzed. In addition, the information from change impact analysis using FCA can still be extracted without displaying a lattice to the user. The result from analysis can be alternatively presented in text format such as list of computations on variables or statements potentially affected by a change.

Separate sublattices that are only connected to the top and bottom element of the lattice become component candidates. In practice, however, due to violations of the information hiding principle, the lattice of concepts has many interferences. Several heuristics to detect atomic components within this lattice despite of interferences have been proposed. Another drawback of this approach is that it may take exponential time to find a concept lattice in the worst case. Practical experiences have indicated that it still takes cubic time on average. Research in atomic component detection has mostly concentrated on the automatic techniques. This has lead to many interesting, yet isolated techniques. A notable exception is Canfora et al. who have proposed to combine concept analysis with other heuristics in order to simplify the concept lattice. However, the other heuristics are only subordinated to concept analysis in their approach. Moreover, little attention has been paid to the question how a maintainer could be integrated into the detection process.

2.3.4.2 Other clustering and grouping techniques [KOS99]

Data analysis underlies many computing applications, either in a design phase or as part of their on-line operations. Data analysis procedures can be dichotomized as either exploratory or confirmatory, based on the availability of appropriate models for the data sources, but a key element in both types of procedures (whether for hypothesis formation or decision-making) is the grouping, or classification of measurements based on either (i) goodness-of-fit to a postulated model, or (ii) natural groupings (clustering) revealed through analysis. The variety of techniques for representing data, measuring proximity (similarity) between data elements, and grouping data elements has produced a rich and often confusing assortment of clustering methods. It is important to understand the difference between clustering (unsupervised classification) and discriminant analysis (supervised classification). The term "clustering" is used in several research communities to describe methods for grouping of unlabeled data. These communities have different terminologies and assumptions for the components of the clustering process and the context in which clustering is used.

2.3.4.3 Feature extraction

Function Extraction deals with the semantics of software behavior. All levels of abstraction in the development of software systems deal with behavioral semantics, from low-level machine language operations to high-level system capabilities. As software systems are developed and evolve over time, semantic content is continuously created, intentionally or unintentionally, correctly or incorrectly. Effective development and evolution of a system depends on how well its behavior is understood by its developers. The complexity and quantity of semantic information can overwhelm developers, leading to loss of intellectual control.
A feature can be described as:

A feature is a group of individual requirements that describes a unit of functionality with respect to a specific point of view relative to a software development life cycle.

This definition is rooted in the problem domain, but shows how a feature can be used in software evolution. For example, a system might support a feature that performs complex calculations in batch mode, without user interaction. To an end-user, this feature is a time saver, because input can be stored in a file or a database to be used at a later time. At the same time, testers might employ this feature to enable regression testing between two versions of the system. Developers might design a specific set of modules to process user input without user interaction to analyze code coverage. A code-profiling tool executing regression test cases exercising that feature can locate the *feature implementation*, and evolution of that feature can commence.

Table 13: Feature/Function Relationships

Feature	*Functions*	*Critical Evolution Viewpoint*
1	Many	Solution domain
Many	1	Problem domain
1	1	None exists
Many	Many	N/A – Must be decomposed

Not every feature is evolved during system evolution, nor should each feature be encapsulated in a fine-grained component. If a feature can be used in another system, its implementation becomes a candidate for reuse. From this candidate set, the organization must still select specific features to evolve. Once the features are associated with their test cases, one can group these features to be evolved with the related test cases for code coverage. The test cases used in this step then can be viewed as the representation of underlying abstract data model.

Responding to external pressures, developers often bypass standard processes to meet project deadlines. This results in inferior coding, such as adding a global variable when one is not required. The internal evolutionary pressures force the developers to either restructure or refactor their code so the future enhancement or maintenance becomes manageable and cost effective. During such evolution, the code is refactored, and protocols and standards are re-established. The end-user may or may not see the changes made to the system, but the goal of such refactoring is to reduce future maintenance costs. Researchers have long identified features as a natural organization of the problem domain. Surprisingly, few approaches in the research literature concentrate on feature-based organization of a system's functionality. In contrast, the solution domain is full of research that incorporates software artifact management activities such as design, component construction, and testing. Regression test suites are an untapped resource for software evolution because they tell a legacy system's story in a way that can be used to identify features of interest to end-users. This information can be used to identify code associated with features, extract that code, and create fine-grained components [MET00, MET05, and KOS99]. These components are inserted back into the legacy system to validate results in two ways. First, one can match the output of the regression tests after the insertion with original outputs. Second, one can measure the cost of adding a new feature and compare that to the prior costs. The typical steps (Figure 21) involved in such a process are:

- Step 1: Select test cases by considering features.
- Step 2: Execute selected test cases using code profilers to locate source code that implements features.
- Step 3: Analyze and refactor source code to create components.
- Step 4: Compare pre- and post-evolution maintenance costs.

However, this approach is based on three basic assumptions. First, it has to be assumed that the legacy system to be evolved is written using a modern programming language such as Visual Basic, C++, Java, or COBOL. This allows the use of existing code-profiling tools to trace the source code implementing a specific feature. Secondly, it has to be assumed that the legacy system has regression test suites. Third, one has to assume that both domain knowledge and expertise for the application and problem domain are available.

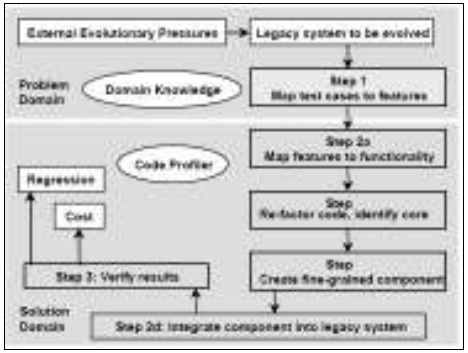


Figure 21 Evolving legacy system features [KOS99]

End-users often view a system in terms of its provided features. They exercise the system features through user input (stored in files or databases) that is often used for system maintenance as part of regression testing. Intuitively, a feature is an identifiable unit of system functionality from the user's perspective. Examples of features include the ability of a word processor to spell check or ability of an accounting system to generate a balance sheet statement for

a given fiscal year. Software developers are expected to translate such feature-oriented requests into system design. *Feature Engineering* addresses the understanding of features in software systems and defines mechanisms for carrying a feature from the problem domain into the solution domain.

- **Domain Knowledge**: There is no substitute for domain knowledge in legacy systems. Through using domain knowledge, it is possible to identify test cases that represent a particular feature or a group of features. It is also possible to construct test cases from scratch to exercise a feature.
- **Documentation**: Legacy systems also have rich regression test suites that consist of hundreds of test cases. In some cases, test suites are well documented and are already grouped by the functionality that needs to be tested.
- **Clustering and textual pattern analysis:** Based on test cases, one can exercise closely related features.. Clustering analysis can therefore be describes as an organization of a collection of patterns (usually represented as a vector of measurements or a point in multidimensional space) into similarity clusters.

2.3.5 Aspect orientation

The general goal of aspect-oriented programming is to extract a functionality that is scattered throughout the whole application such as logging, transaction management, or security management into a separate module. The same approach can also be used to capture components and features, easing the component substitution process. Depending on the application domain, other aspects can be identified as well [KVA05]. AspectJ is a compiler that supports aspect-oriented programming. AspectJ is an extension of the Java programming language that is fully backward compatible to the Java language specification. In addition to the functionality provided by Java, AspectJ supports the definition of aspects. One has used AspectJ to define a set of instrumentation aspects that add code to a given Java program to gather enough tracing information such that the program can be reverse engineered. Within Aspect instrumentation, one uses this variable to obtain the method signatures and arguments. This allows engineers to track the function calls and the objects that are passed around by a software application Aspect simply forwards each method invocation to one or more recorder modules. If necessary, the instrumentation aspect may be changed for instance to instrument only a part of the original application. Using AspectJ to obtain the trace-data from the application to be understood has the following advantages. The aspect is woven into the program autonomously. AspectJ does not turn off Java's just-in-time compiler as has to be done for instrumentation approaches using Java's Virtual Machine Debug Interface (JVMDI) It provides a mapping from the joinpoints specified to the application's source files and line numbers. AspectJ allows accessing the parameters that are passed as part of a function call and thus allows the traces to be tuned to specific object instances. A minor drawback of AspectJ, however, is that the aspect woven into the application remains unchanged for the whole program execution. This is why the aspect generates data for each method invocation, and subsequently, is processed by a recorder module that is responsible for filtering the method invocations the engineer is interested in. The recorder, however, can be customized during run-time allowing engineers to enable tracing only during the execution of those functions, objects, or object instances, he is interested in. Since the current version of AspectJ uses the source code of an application to weave in the aspects, it requires the availability of the applications source code. Starting with AspectJ Version 1.2. addresses this limitation by being able to operate directly on the application's byte code without the need for the application's source code.

2.3.6 Data Reverse Engineering [DEM99]

Data reverse engineering is an aspect of reverse engineering, which is sometimes misunderstood. Data reverse engineering is a valuable and essential part of reverse engineering. It gained on momentum during the Y2K problem. This was specifically due to the particular need for identifying and modifying these parts of the systems that are affected by the Y2K problem. Data reverse engineering techniques consist of a more restrictive subset than those used in reverse engineering. As Elliot Chikofsky stated in his preface to the book *Data Reverse Engineering: Slaying the Legacy Dragon* [CHIK96], "Reverse engineering is a process to achieve understanding of the structure and interrelationships of a subject system. It is the goal of reverse engineering to create representations that document the subject and facilitate our understanding of it – what it is, how it works, and how it does not work. As a process, reverse engineering can be applied to each of the three principal aspects of a system: data, process, and control. Data reverse engineering concentrates on the data aspect of the system, i.e., the organization. It is a collection of methods and tools to help an organization determine the structure, function, and meaning of its data." It is the restriction of data reverse engineering to the data portion of a software system that makes it a complex and

most interesting activity. It has to be noted that database schemas only specify the constraints allowed by the underlying database system and model. However, the problem domain may involve other constraints not expressed in the schema. By inspecting samples of the actual data stored in the database you can infer other constraints.

A typically data reverse engineering environment is based on techniques from the areas of data mining, pattern matching, and clustering. Users define a graph-based architectural pattern of system modules (subsystems) and their interactions based on domain knowledge, system documents, that are supported through clustering techniques. Using an iterative recovery process, the user constrains the architectural pattern, and the tool provides a decomposition of the system's entities into modules or subsystems that satisfy the constraints. Extracted information is stored in some form of a *database* to make its data persistent by providing a static description of the data inside the database. Data reverse engineering requires some *expertise* with mapping data-structures from your implementation language onto a database schema, enough to reconstruct a class model from the database schema. It also typically requires some general understanding of the system's functionality and the context the data is used.

During a typical data reverse engineering task, one checks first the entities that are stored in the database, as these most likely represent valuable objects. A class model representing those entities is derived, to document the knowledge. A more detailed scenario based on a relational database is given below. It should be noted that steps 1-3 are quite mechanical and can be automated quite easily.

1. Collect all table names and build a class model, where each table name corresponds to a class name.
2. For each table, collect all column names and add these as attributes to the corresponding class.
3. Collect all foreign keys relationships between tables and draw an association between the corresponding classes. (If the foreign key relationships are not maintained explicitly in the database schema, then you may infer these from column types and naming conventions.)
4. After the above steps, a class model can be created that represents the entities being stored in the relational database. However, because relational databases cannot represent inheritance relationships, there is still some cleaning up to do.
5. Check tables where the primary key also serves as a foreign key to another table, as this may be a "one to one" representation of an inheritance relationship inside a relational database. Examine the SELECT statements that are executed against these tables to see whether they usually involve a join over this foreign key. If this is the case, transform the Association that corresponds with the foreign key into an inheritance relationship.
6. Check tables with common sets of column definitions, as these probably indicate a situation where the class hierarchy is "rolled down" into several tables, each table representing one concrete class. Define a common superclass for each cluster of duplicated column definitions and move the corresponding attributes inside the new class. To name the newly created classes, one can use your imagination, or better, check the source code for an applicable name.
7. Check tables with many columns and lots of optional attributes as these may indicate a situation where a complete class hierarchy is "rolled up" in a single table. If you have found such a table, examine all the SELECT statements that are executed against this table. If these SELECT statements explicitly request for subsets of the columns, then you may break this one class into several classes depending on the subsets requested. When you have incorporated the inheritance relationships, consider to improve the class model exploiting the presence of the legacy system as a source of information. In particular you should inspect data samples to check for missing constraints and you should check at which queries are executed against the database engine to infer missing foreign keys.

Advantages of this approach can be found in the following areas:

- **Team communication.** Capturing the database schema, allows the improvement of the communication within a reverse engineering team and maintainers. Moreover, any if not all of the people associated with the project will be reassured by the fact that the data schema is present, due to the fact that many development and maintenance methodologies stress the importance of the availability of data and schemas.
- **Model of critical information**. The database usually contains the critical data, hence the need to model it because whatever future steps you take you should guarantee that this critical data is maintained.

However, there exist also some major limitations with data reverse engineering:

- **Limited Scope.** Although databases are crucial in many of today's software systems, they typically only involve but a fraction of the complete system. Therefore, one can not rely on this information as sole source to comprehend the internal aspects of a system.

- **Requires Database Expertise.** Data reverse engineering requires a good knowledge of the underlying database plus structures to map the database schema into the implementation language. Therefore, data reverse engineering should be done by people with both, database and implementation language expertise.
- **Polluted Database Schema.** The database schema itself is not always the best source of information to reconstruct a class model for the valuable objects. Many projects must optimize database access and as such often sacrifice a clean database schema. Also, the database schema itself evolves over time, and as such will slowly deteriorate. Therefore, it is quite important to refine the class model using data sampling and run-time inspection.
- **Run-time inspection.** Tables in a relational database schema are linked via foreign keys. However, it is sometimes the case that some tables are always accessed together, even if there is no explicit foreign key. Therefore, it is a good idea to check at run-time which queries are executed against the database engine.

2.3.7 Visualization

It is commonly accepted that software systems have grown too large to be statically verified and analyzed. This is true even when the software is decomposed into well-defined software components. Until the software engineering community develops more powerful analysis techniques, there is a need for developers to assess the run-time behavior of complex software systems. There exists a large body of work on software visualization. Some of these visualization techniques were already covered in a previous state of the art report on architecture recovery. With the majority of the work focusing on static structural representations, e.g. class models, package diagrams, there exists a need to address open issues that are closer related to the visualization of dynamic aspects as well as considering different visualization metaphors. Furthermore, there is also a need to support the visualization of the information extracted from the various analysis techniques. These results often do not conform directly to the more traditional representation. As in other disciplines of scientific visualization, it makes sense to combine analysis techniques with more sophisticated visualization techniques to assist the user in comprehending data and consequently, to identify patterns/relationships that would be otherwise not be as obvious. With an increase of the abstraction level, there is also a need to incorporate not only analysis information but also prior domain expertise into the visual representation. In particular, there should be a focus on closing the conceptual gap between reverse engineering and forward engineering. Forward engineering typically incorporates both domain expertise and application expertise of the designer and requirements engineer. The section does not discuss any specific visualization techniques but rather focus on a brief discussion of the major challenges associated with closing this conceptual gap. These challenges can be briefly summarized as follows:

- Identifying logical groups and visualizes them to support the comprehension process. E.g. features, aspects or any other logical grouping that was identified as part of the analysis step.
- There is also the challenge of recreating a visual that closely matches the representation that was originally created during the forward documentation or as part of any existing documentation.
- Scalability, navigability, and representing the information in such a way that it matches closely a maintainer's mental model and expertise.
- Finding (new) metaphors which provide views and represent expertise, domain knowledge and source code related information in a fashion that support and benefits the comprehension process.

2.3.8 Discussion – FOSS component substitution at the feature/package level

Information needs:

The majority of the techniques surveyed to support component substitution at the feature/package level require some types of static program models as the underlying basis for the analysis. These models are often complemented by dynamic information that is captured in execution traces. Depending on the specific technique, there might also be a need for other documentation, e.g. design, requirement, specification and testing documents. Common to most of the surveyed techniques is, that a domain and/or application expert has to be on hand to provide input to the analysis techniques.

Techniques:

The techniques reviewed in this section can be grouped in three major categories.

- Feature/grouping extraction techniques including concept analysis, clustering techniques, etc., that help to identify and recover components in the systems. The identification and scope of components provides the basis for the component substitution.
- Dependency analysis techniques in the form of ripple, change impact and traceability analysis techniques. The general goal of these techniques is to correlate the change to the rest of the system, by identifying what parts of the system (and its documentation) might be affected by the change (component substitution).
- Visualization – providing high level views of the static program model, incorporating clustering and grouping techniques.

Information provided and applicability to component substitution:

Several techniques provide potential candidates for such component identification (including concept analysis, trace analysis, scenario analysis). After the identification, the change (component substitution) can be performed. The challenge lies in identifying what are the affects of the substitution and which parts of the system have to be retested to verify that the substitution process did not cause any undesirable side-effects. Techniques to support this phase of the component substitution include ripple effect, change impact analysis and dependency analysis. The challenges with the source code based impact analysis and dependency analysis approaches is the need that the maintainer has to have a good understanding of the system, has to be able to located the component and perform the substitution to be able to determine the impact of the modification on the overall system. From a more pragmatic point of view however, this process might be easier at the package or feature level, due to the possible existence of well defined interfaces that restrict the impact of such a change and make the location of the parts to be substituted easier.

The existing visualization approaches are currently focusing mainly on the creation of structural views of the system and its components. These views do typically not incorporate domain or application expertise a programmer might have. Also, their visual metaphor might be a restricting factor, limiting their expressiveness, scalability and therefore their applicability during the substitution process. There is a clear need in identifying and supporting meaningful clustering and grouping techniques to support the maintainers and to help close the conceptual gap that might exist between the visual representations and the mental model a maintainers has of the system to be analyzed.

Table 14 Applicability of FOSS analysis at the feature/package level

Technique	Application
Change impact analysis, ripple effect, traceability, dependency analysis	Comprehension, change impact analysis, maintenance, protocol verification
Concept analysis, grouping techniques, feature extraction	Comprehension, testing, refactoring, feature identification and analysis
Behavioral analysis	Comprehension, component interaction analysis,
Aspect oriented	Refactoring, tracing, comprehension, maintainability
Data reverse engineering	Program comprehension, maintenance, component identification and grouping
Software Visualization	Structural comprehension, Visual representation that closes the conceptual between software, maintainers expertise and domain knowledge.

2.4 Subsystem/Architectural level

Software architecture plays a vital role in the development of large software systems. For the purpose of maintenance, an up-to-date explicit description of the software architecture is needed to support the understanding and comprehension of it. However, many large complex systems are not well documented or have no up-to-date software architecture documentation. Particularly in cases where these systems have a long lifetime, the (natural) turnover of personnel will make it very likely that many employees contributing to previous generations of the system are no longer available. A need to 'recover' the software architecture of the system may become prevalent, facilitating the understanding of the system, providing ways to improve its maintainability and quality and to control architectural changes and their impact. Furthermore, component substitution at the architectural (or subsystem) level can only succeed if the maintainer has a clear understanding of the interconnections of the different system parts and how these parts work together to achieve specific goals. Therefore, for a software maintainer to be able to substitute components or parts of system, it is necessary that one has an understanding of the dependencies, the deployment of functionality across the system.

During forward engineering, a typical view generated is the 4+1 view. One of the major goals of most architectural reconstruction and recovery approaches is to recreate such a 4+1 view, with a *view* being a representation of a whole system from the perspective of a related set of concerns. While it is now generally accepted that the architecture description should be composed of multiple views, the terminology related to views is not yet widely accepted. In this paper, one refers to the IEEE 1471 standard [IEE00]. In IEEE 1471, a view conforms to a *viewpoint*. While a view describes a particular system, a viewpoint describes the rules and conventions used to create, depict, and analyze a view based on this viewpoint. A viewpoint specifies the kind of information that can be put in a view. For architecture reconstruction, multiple viewpoints and views are beneficial. Different viewpoints help the architect determine what information should be reconstructed in order to solve the problem.

Despite the variety of models, there are recurring architectural parts that constitute models that have been discussed in the community over the past several years, such as patterns and their detection in existing systems, architecture styles, and quality attribute models. Furthermore, there is a rich set of strategies and techniques available to collapse detailed source information into more abstract structures. It is important to obtain an initial concept (hypothesis) for the model to guide the identification of:

- facts from the existing system;
- useful collapsing strategies;
- potential patterns;
- relevant quality attribute models.

Architecture reconstruction is hard to achieve without any initial idea about the architecture of the existing system. It should also be pointed out that at this abstraction level, software component substitution requires a combination of various approaches and techniques at different levels of granularity. The techniques include architectural recovery, integration of analysis techniques introduced in sections 2.1 -2.3, and the involvement of domain and application experts.

2.4.1 COTS

Component-oriented programming should facilitate building software like a puzzle, whose parts (components) are created by mainly third party providers.. From a technical view point, it is difficult to identify techniques specific to the COTS substitution at the subsystem level. A combination of the techniques discussed in the previous sections will have to be re-applied to support component substitution at this higher-level of abstraction. At the level of subsystems, the component substitution process will have to focus not only on technical but also on managerial issues related to a substitution. One of the major challenges is to identify the component that has to be substituted and to select a new component that meets best the new requirements. These questions lead to the problem of comparing a component with a need. This is not a new problem or problem specific to component substitution. The same problem was occurring in previous programming paradigms, such as in object-oriented when people were working on software reuse. In this case, a part of the problem was solved by a good use of mechanisms such as typing or sub-typing. However, in the COTS context, easy software substitution or reuse does not correspond to any

typing relationship. Indeed, the selection of a COTS is based on its offered services and not on its type. This enforces the "black box" aspect of the COTS that requires that not only their functional, but also their non-functional properties are well described and defined. These techniques are at the current stage not covered in this report, since it focuses mainly on the substitution process itself, rather than on the pre-requisite stages.

Table 15 Overview COTS analysis at the subsystem/architectural level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Testing based	Test suites, traces	Functional and non-function properties, e.g. security	Comprehension, impact analysis	N	Y
Comprehension of standard COTS	EFSM system model Input testing	State change, dependency analysis with libraries, etc	Verification, testing, comprehension	Y	Y

2.4.1.1 Testing based comprehension

Testing can be applied to evaluate the interactions between the component and the rest of the system against application requirements. Non-functional properties such as security and fault tolerance are also evaluated. Since component code is usually unavailable, testers resort to black-box testing. In [KOR03], Korel deals with testing COTS components, by utilizing an operational profile. The work proposes an interface probing technique to understand black-box components, where the tester defines specifications for the required component behavior. The approach automatically generates *check-code* with the required behavior [HIS99]. Another approach to comprehend components and their interactions with other subsystems is interface probing. A developer has to design a set of test cases that are executed on the component. The results can be evaluated and interface probing can be applied iteratively. While black-box understanding methods can be applied when no source code is available they are fairly limited in their ability to provide detail insights of the internal behavior of COTS.

2.4.1.2 Comprehension of standard COTS frameworks

For the comprehension of standard COTS frameworks [DON05], like COM+, a *test case generator* can be used to generate test cases using black-box techniques. The goal of COM+ analysis process is the generation of an abstract model describing important architectural characteristics of each COM+ component. A list of architectural characteristics is described below for COM+ component with their different information sources including [PIN03]:

- *Source code* comprises definition and source files that define and implement the interfaces of COM+ components. Source files also contain the statements that indicate certain architectural characteristics such as for example SetComplete() and SetAbort() for handling transactions.
- *Type libraries* correspond to interface definition files and provide detailed information about the interfaces implemented and provided by COM+ components. Instead of parsing IDL files, we use the COM+ API to extract the type library information of each COM+ component. In particular, such an approach is useful if no IDL files are used or have been created automatically by the IDE as it is in the case in VisualBasic.
- *Registry*: COM+ allows the configuration of component behavior at deployment time such as transaction semantic and security settings. This information gives a rough estimate about deployment specific architectural characteristics of COM+ components and complements the source code and type library analysis results.

Component models provide also access to Meta data to aid system integration of components developed independently. Meta data contains descriptions about external visible interfaces and the corresponding methods that are exposed externally to clients. COM+ stores Meta data in type libraries. These libraries are either stored in additional files with suffix *.tlb* or directly placed in the component's image file. In both cases the COM+ API function `LoadTypeLib` may be used to load a type library. These API functions return a COM+ component that implements the predefined `ITypeLib` interface and provides methods to retrieve the `ITypeInfo` interface that contains the necessary functionality to access COM+ meta data about interfaces, methods, parameters, user-defined types, enumerations and all names used for defining this meta data.

2.4.2 FOSS

Often, software developers are expected to maintain poorly understood legacy systems. Unfortunately, due to the lack of proper understanding of the system, any extensions or modifications will lead typically to a degradation of the system structure, its maintainability and comprehensibility. Specifically, each modification moves the structure of the system away from its original design, making maintenance increasingly difficult and if such systems are to survive, they need to be modified, reengineered or substituted. Component recovery supports program understanding, architecture recovery, and re-use.

Table 16 Overview FOSS analysis at the subsystem/architectural level

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Architecture extraction	Source code, domain knowledge, file names	Interaction among components	Comprehension, dependency analysis	Y	N
Architecture Reconstruction	Program model, source code, domain knowledge, scoping, visualization	High-level views	Validating of performed substitution	Y	N
Architectural Transformation	Program model task, domain knowledge, trace information	Architectural analysis, transformed architecture	Adaptation of architecture, legacy systems conversion	Y	N/Y
Architectural metrics	Program model, CVS data, package information,	Object/package dependencies	Evaluation, comprehension	Y	N
Visualization techniques	Program model, file, package information	Abstract view, package, 4+1 view	comprehension, impact analysis	Y	N
Impact analysis	Documentation, domain knowledge, program model, query language	Affected subsystem parts	Dependency analysis	Y	N
Dynamic analysis, logging, visualization	Log files, sampling, traces	Communication dependencies, interaction diagrams, behavioral models	Re-modularization, interactions analysis, comprehension	**N**	Y

2.4.2.1 Architecture extraction

One approach to extract knowledge from an existing system is to use architecture extraction or recovery. The goal is to identify the architecture that corresponds to an existing (implemented) software system. That is, an organization has an implemented system that was designed and implemented in the absence of the notion of software architecture, but now wishes to construct an architectural description of the software in order to facilitate the maintenance of the system as well as leverage reuse of the system or its subsystems [KOS06]. One has found this to be a very common need among industry and government. Architectural extraction is also a very effective way to train developers to design software at the architectural level. Starting with an existing system and "reverse engineering" back to the architectural level is an excellent exercise in discovery of the similarities and differences in an architectural description versus a traditional detailed design view of the software. Of course, defining the architecture of an existing system is not an easy task. The assumption is that there exists a design specification of the system, in a particular design notation (i.e. textual, graphical, hybrid). The goal of architectural extraction is the definition of the collection of architectural-level components, connections, constraints, and associated rationale that describes a specific system. The architectural extraction should include all necessary views of the system. That is, there should be a user view, a developer view, a maintenance view, a testing view and so on. Each of these views may in turn require their own sub-views. Typically, these techniques determine how low-level components interact. However, just as important is the need to determine the system hierarchy: how are the modules grouped into subsystems and how are the subsystems grouped into higher level subsystems? This hierarchy or decomposition can be determined from file naming conventions, directory information, program structure information, interviewing people familiar with the software, etc. It is generally accepted that the component interactions (including program level dependencies such as calls from procedure to procedure) together with the system hierarchy, define the software's structure or architecture. Existing tools extract facts from source code and use them to visualize how

components such as files/modules interact. For large software systems, the graph will be huge (often containing hundreds of thousands of edges). Hence, directly viewing such a graph is of no help. During *architecture understanding*, one needs to describe the module interactions at higher-levels of abstraction (e.g., at the top subsystem level) and also, one needs to be able to simplify this information to produce various architectural views.

2.4.2.2 Architecture Reconstruction [DEU05]

Architecture reconstruction in practice has been predictably ad-hoc, using simple tools and a large amount of manual interpretation. Researchers have been trying to improve the state of the practice primarily by providing better techniques and tools (e.g., cluster or concept analysis, program analysis, and software visualization). The application of these techniques usually involves three steps: extract raw data from the source, apply the appropriate abstraction technique, and present or visualize the information obtained [DEU04].

Software architecture reconstruction is a special form of software reverse engineering. Many reverse engineering approaches are based on an extract–abstract–present cycle, in which sources are analyzed in order to populate a repository, which is queried in order to yield abstract system representations, which are then presented in a suitable interactive form to the software engineer. Software architecture reconstruction is the process of obtaining a documented architecture for an existing system. Although such a reconstruction can make use of any possible resource (such as available documentation, stakeholder interviews, domain knowledge), the most reliable source of information is the system itself, either via its source code or via traces obtained from executing the system.

2.4.2.3 Architectural Transformations

One commonly used approach to architectural transformation is graph rewriting .A graph model of the system is extracted, to support its exploration. This model can then be used to provide guidance during the structural understanding of a system and allow creating views based on this recovered structure. Furthermore, the graph can then be used to apply different types of transformation to guide during the analysis and comprehension of the system and its interconnections [KAZ96, KRI99, and CAR99].

Transformations for analysis. Transformation for analysis can be applied to discover various kinds of information about the software system. For example, what modules interact in a cyclic pattern. This kind of information is commonly used to determine steps to be applied for system modifications.

Transformations for modification. These transformations can be applied to change the system structure. For example, one may identify unexpected interactions between subsystem V and W and by moving certain modules one can eliminate these interactions.

Questions to be answered by transformations in this section focus on *architecture analysis*, during which one discovers various kinds of information about the system that can help restructuring or modifying an architecture. Questions like, "How are the concrete and conceptual architectures different and what has caused the inconsistencies?" or "Which modules should be made local to other modules?" or "Which modules exhibit poor information-hiding?" need to be answered so that one can decide what should be changed and help identifying the change and how and what type of substitution should be performed.

The first step in the transformation of code and software architecture is to reconstruct the existing software architecture. This is done by extracting architecturally important features from its code and employing architectural reconstruction rules to aggregate the extracted (low-level) information into an architectural representation. A system's architecture can be extracted through static analysis of the source code and the system's makefiles to identify function-call and built-from relations. The former identifies function calls within the system as well as calls to the socket primitives used for communication between client and server. The latter identifies how the executables in the system (one for each for the client and server) are built from the source files. In addition, a dynamic extraction based on collected trace information can be performed to determine the run-time connectivity of the system as a whole, e.g. to resolve issues related to distributed and run-time behavior.

This combination of static and dynamic analysis provides often sufficient information to reconstruct the run-time connectivity of the architecture. Once the architecture has been extracted, one can view the components and connectors of the system as possessing architecturally significant properties. Possible features of architectural elements, both components and connectors, can be divided into information that can be derived from a temporal

perspective and information that can be derived from a static perspective of an architectural element (Figure 22). The static view's features are enumerated below as a set of categories and an explanation of the reasoning behind each category [CAR99]:

- *Data Scope:* What is the largest scope across which data can be passed by the element? Possible answers are that the element passes data within a virtual address space, across virtual address spaces but within a single physical address space, or across a network.
- *Control Scope*: What is the largest scope across which control can be passed by this element?
- *Transforms Data:* Are the element's outputs a transformation of its inputs?
- *Binding Time*: When are the sources and sinks of an element bound? Possible answers are at specification time, invocation time, or execution time.
- *Blocks:* Does this element suspend when it transfers control to another element?
- *Relinquish:* If this element has a thread of control, does it ever voluntarily relinquish it?

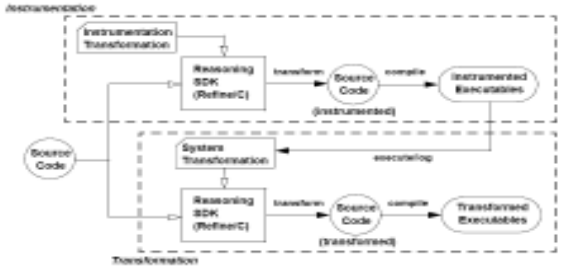


Figure 22: Summary of Instrumentation and Transformation [CAR99]

2.4.2.4 Architectural Metrics [GOR05]

Architecture reconstruction and evaluation should be iterative and interactive. Scoping should go hand in hand with the inspection of high level design metrics and identification of anti-patterns. Many reverse engineering tools focus on increasing understanding of the system based on standard views of the software architecture. Without additional metrics, these views only provide limited visual cues on where to look for potential problems. The underlying metric of Delta-IC is extended towards abstract data types that can also be detected. Furthermore, the connectivity metric of Delta-IC is combined with a cohesion metric based on vertex connectivity. Among the many metrics, the following proved to be particularly useful for architecture reconstruction and evaluation:

- **Afferent Couplings (Ca) of Package A:** The number of other packages that depend upon classes within package A. This is an indicator of package A's responsibility.
- **Efferent Couplings (Ce) of Package A:** The number of other packages that the classes in package A depend upon. This is an indicator of package A's independence.
- **Abstractness (A) of Package A:** The ratio of the number of abstract classes and interfaces in package A to the total number of classes and interfaces in package A.
- **Instability (I) of Package A**: I = Ce/(Ce+Ca). This is an indicator of package A's relative *resilience to change.*
- **Distance from the Main Sequence (D):** The perpendicular distance of a package from the idealized line A+I = 1. This is an indicator of the package's balance between abstractness and stability. In an ideal situation, if the package is almost completely abstract (A->1), it should be very stable (I -> 0). On the other hand, if the package is almost completely concrete (A->0), its instability (I->1) could be justified.

The overall stability of the system can then be calculated by first determining the potential average transitive impact AI (Average Impact) of each component on the rest of the system. The overall Average Impact is then the average impact of each component. Transitive impact analysis is different from the standard coupling measurement and appears to be more useful in architecture evaluation. This is because standard coupling metrics only capture the

impact of changes to adjacent architectural elements. However, whether the impacts of a change have the potential to be contained or propagated can not be seen directly.

Based on the transitive change impacts, the following design metrics and anti-patterns can be specified:

- **Global Butterfly:** If the component is changed, it may affect many other components.
- **Global Breakable:** The component is often affected if anything in the system is changed.
- **Global Hub:** The component is both a global butterfly and a global breakable.
- **Skeleton:** This layered view of the system is constructed by putting objects (class/interface/package) that do not depend on anything at the bottom of the visualization. The objects that are dependent on the lowest layer appear in the above layer, and so on. In this view, a stable system should have a normal pyramid shape. An unstable system may look like an upside down pyramid shape. The skeleton can be cross-referenced with other artifacts, like packages or inheritance trees.

These design metrics and anti-patterns are proposed to be capable of focusing a maintainer's attention on certain components in the architecture. However, these metrics cannot help in further reconstruction and deeper analysis of these components. In the field of Component-Based software engineering, the future is pictured where more and more software is built from *components*, meaning binary executables (EXEs, DLLs, etc.) possibly developed out-of-house. Therefore, to make component-based systems more maintainable, one must be prepared to measure and estimate maintainability on the architectural level when source code is no longer available.

Other Measurements

There exist other source code based metrics at the architectural level include counting dependencies between components at the subsystem level; "fan-in" and "fan-out"; number of calls in, number of calls out. Also the work of Chidamber and Kemerer was extended, by refining these existing measurements to include component specific measures. Examples of these measurements include: *number of children for a component* (NOCC), level of coupling for a component: *external CBO* (EXTCBO) which corresponds to the number of external classes coupled to it and the *response set of a component* (RFCOM) is the number of all the methods in the member classes and the methods called by those classes.

2.4.2.5 Visualization techniques

The challenge in the visualization of complex software systems lies in the undistorted and comprehensible representation of the large amount of extracted data. This implies two goals: First, the information density in the views should be maximized under the constraint of comprehensibility. However, even with a large information density, not all details of the data can be represented in one view. Thus the second goal is to provide intuitive navigation, which allows the user to move easily between views at different levels of abstraction and for different parts of the visualized system. Clearly, there is a trade-off between information density and comprehensibility. Many three-dimensional visualizations have a high information density, but appear cluttered, occlude distant information, and provide no global overview. On the other hand, many two-dimensional visualizations are very clear but reveal little information. Therefore, it is desirable to find a compromise between information density and clarity.

Also the support for an adequate navigation through the visual representations plays an important role. There are two *navigation styles* for investigating information spaces: searching and browsing. *Searching*, sometimes called analytical searching, is a planned activity with a specific goal, such as to find a particular fact. It is often associated with who, what, when, and where questions. Searches involve formulating queries or looking at indices. In contrast, *browsing* is an exploratory strategy, with no fixed endpoint, and is relatively unstructured. The knowledge seeker relies on serendipity to uncover relevant information. Browsing is associated with why and how questions and exploratory investigations, and involves actions such as flipping through the pages in a book, or following links through hypertext. Software architecture visualization tools tend to support browsing, that is, exploration by following concepts. If architectural diagrams are to be used during daily software maintenance tasks, these tools also need to support specific fact-finding through searching. Searching is essential to program comprehension and hypothesis testing. Furthermore, searching allows users to reverse the abstractions in architectural diagrams and access facts in the underlying program code.

When browsing a software architecture diagram, the user explores a software system via visualization. The visualization is generated by abstracting details from the source code to show a conceptual representation of the system. Since browsing is suitable for exploring new domains, such an interface is appropriate for users who are unfamiliar with the software system. However, if a software maintainer wants to learn about the source code and not just the architecture, he/she will need to access the facts that were used to construct the abstractions. In software

visualization, this process is called *reverse abstraction*, moving from representations of concepts to the underlying facts. For instance, to find the lines of code that is represented by a single edge, the software maintainer needs to reverse abstract the architecture diagram. It is easier to reverse abstract using searching. This navigation strategy is commonly used by programmers through text editors and utilities, such as `grep`. Unfortunately, tools that work with text do not carry over to diagrams.

Currently, the most popular graphical modeling language in software engineering is UML. In UML, the static structure of a system is modeled by class diagrams. Classes can be grouped to packages to obtain diagrams at a higher level of abstraction. UML, like other diagram notations, includes no advanced graphics and visualization techniques. On the one hand, this facilitates the drawing of diagrams by humans and the representation of diagrams in paper documents. However, it also decreases information density and control over the level of abstraction, which limits the scalability of the UML representation. An inherent difficulty in general software visualization, whether static or dynamic, is the lack of an agreed upon set of "ideal" metrics for component-based software systems.

. For the facilitation of maintenance tasks, project managers and developers often turn to the evolution history of the software system to recover various kinds of useful information such as anomalous phenomena and lost design decisions. Mining information from the evolution history is difficult and time consuming due to the presence of abundant data. A variety of evolution visualization techniques have been used to cope with the large amount of details present in the evolutionary process of software systems. These visualization techniques can be effective for uncovering information deeply buried in history data such as source releases, bug reports, and maintenance logs. The uncovered information can be useful for understanding software system evolution and answering questions like the following: Which parts of the system appear unstable over long periods of time? Which subsystems are the most likely to have a fault? What code did a programmer modify and how? This section describes a software evolution metaphor based on spectrographs by drawing an analogy to sound spectrographs.

Spectrograph Dimensions [WUH04]

A sound spectrograph provides a visual representation of the frequency content of sound and its variation in time. It is normally presented in the form of an XY graph, in which the horizontal axis X denotes the time dimension, and the vertical axis Y denotes the frequency range (Figure 23). The brightness of a position in the graph indicates the relative amplitude of the energy present for a given frequency and time.

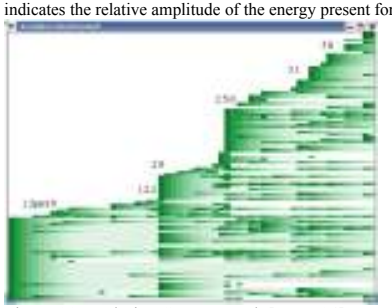


Figure 23 Evolution Spectrograph [WUH04]

Analogous to sounds, software evolution can be characterized in terms of *time*, *spectrum*, and *measurement*, and visualized using spectrographs. It is based on the following three dimensions

Time

The time dimension denotes the whole or partial lifetime of a software system. Time can be measured in two ways. First, we can measure time in units of evolution events, such as software releases and repository commits. Second, one can use fixed-length periods as time units, such as months.

Spectrum

Analogous to sound decomposition into frequency components, software system decomposition into smaller software units provides a measurement basis for the Y axis. In the spectrum of sound, frequency components are

arranged into an order according to their values. Similarly, software units must be ordered by a particular property. A software system can be decomposed at varying levels of granularity, such as the subsystem or file level. Such a structural decomposition is one example for a spectrum, others include programming language, developer, etc.

Measurement

For a component in the spectrum, one can measure a particular aspect or property of that component at any points during the lifetime of a software system. A variety of software metrics, such as Lines of Code (LOC), Fan In/Out of dependencies, and defect density can be computed on a per unit basis.

2.4.2.6 Software Architecture Impact analysis [LAS02]

For architectural impact analysis there exist techniques that are based on different information resources. They can vary from documentation to graph theory models. Based on the source code and a particular change to that source code, Kung et al propose a way to do automatic ripple effects analysis of the change. Lindvall and Sandahl have reported on a study on the accuracy of the designers in predicting necessary changes. Empirical results on source code impact analysis indicate that software engineers, when doing impact analysis, predict only half of the necessary changes. However, it is noted that the modifications identified are correct. In other studies, it has been shown that about 40% of the changes are visible in the object-oriented design. Some 80% of the changes would become visible, if not only the object oriented design is considered but also the method bodies. These results suggest that impact analysis at the architectural level to be less complete in the sense that not all changes will be detected.

Software architecture impact analysis is concerned with measuring change on an architectural level. This not only identify the parts of the architecture that are affected by a change, but also comparing the before and after situations. During impact analysis, the software architect interacts with the underlying model. If he or she sees possibilities for improvements, changes can be made in the model and the results can be evaluated and compared with the original. Before the software architect can start interacting with the model, information has to be extracted from the software. The extraction is mostly an automated process. Scripts can be used to extract information from the software often in combination with commercial tools. The type of information extracted is for instance the use relation between functions, otherwise known as the call-graph. The part-of relation for each level of abstraction can also be extracted. For example, the part-of relation between units and modules can be extracted by scanning the filenames of the units for their prefix. Simpler is the part-of relation between modules and subsystems, which relates to files in directories. Note that the same module prefix may have been used in different directories, which in the example is considered an architecture violation that needs to be corrected in a first transformation.

One way to evaluate a recovered architecture is using architectural metrics for the different quality aspects. An example of a metric is the cohesion of a unit, which can be expressed as the quotient of the number of internal static function calls and the number of all possible internal static function calls. A metric for coupling defines the rate of external connections. The coupling metric, like the metric for cohesion, can be defined for different levels of abstraction. A metric for layering indicates the rate of up-calls in a layered architectural model. Restricting views and zoom functions can be calculated on the model to let the software architect focus on the problems at hand. Note that architecture impact analysis is a highly interactive process that cannot be completely automated. The final decisions have to be made by the architect, since the process can only show possible changes and their impact.

2.4.2.7 Dynamic comprehension techniques

Software reverse engineering is defined to be a process of analyzing software components and their interrelationships in order obtain a description of the software at a high-level of abstraction. Suggested approaches to reverse engineering include the use of structural re-modularization and architecture recovery. These techniques are primarily static, although some dynamic approaches have been developed. Once a model is obtained, a question of traceability arises with regards to consistency between the original source and the derived model. As the gap between the levels of abstraction of an original source and reverse engineered model increases, this question of consistency becomes increasingly important.

Figure 24 Sample log [GAN03]

A common technique employed by software developers is the use of log files [GAN03] to generate traces of observed software behavior. Several different approaches for creating log files exist including the use of compiler directives at the time of development and post-development instrumentation. As a resource for reverse engineering, a log file has the advantage of being an accurate account of software behavior. However, a disadvantage is that the log file provides only a subset of possible behaviors of the software. As a result, to mitigate the risk of using log files as a source of information for whatever reason, approaches such as static behavior sampling are needed to ensure that a log file trace provides a reasonable or adequate amount of behavioral coverage.

Figure 24 contains a log file generated by a program that is used to control a heating system based on temperature sensor data. This log file represents output that is commonly captured from systems either as debugging output or via event capture systems. In this case, the data was captured using an event capture system that has been developed using the Java Platform Debugger Architecture. Once a log file like the one shown in Figure 29 is generated, it can be used to verify the correctness of a recovered model of an architecture, as long as the architecture is expressed as a state model. In order to use a log file in this manner, its contents (or subsets thereof) must be translated into an equivalent logical form. Dwyer et al. [DWY98]introduced the notion of specification patterns for finite-state verification as a means for specifying and verifying common properties. Properties found in log files can be easily described using many of the patterns described by Dwyer et al. The log file generated from a program can log high-level information such as thread start and stop as well as method calls. The event capture system also has the capability to log object state changes that occur when attributes values are modified. The potential uses for the log file are two-fold. First, the log file can be used to identify whether the system meets certain architectural assumptions that may arise in during model construction. Second, the log file can be used to verify problem specific queries such as presence of mutual exclusion properties.

Mittermeir and Pozewaunig [MIT02] developed a technique called *static behavior sampling* (SBS), which exploits the behavior of static components directly, without taking detour of externally added descriptions. The SBS technique calls on historical data about component executions, which is provided by test data. The data helps infer information about characteristic behavior of components, without having to analyze or execute the component. This approach lays the groundwork for ensuring that a generated log file has reasonable behavioral coverage and the discovery of concurrent behavior patterns from system events traces. The systems' event trace helps determine the concurrent behavior and acts as a major source for deriving a model. The technique is based on probabilistic and statistical analysis of the event traces. This approach is an example of a strategy that would benefit from log file based verification. Andrews [AND98] provides a framework for automatically analyzing log files using state machines in the context of testing. The Log Files are created using logging policies and a standard format which specifies certain keywords. A log file analyzer is specified formally using the *Log File Analysis Language* (LFAL) and is built as a set of parallel state machines with each log file machine checking one thread of event. The approach followed by Andrews uses log files and model checking to perform software testing.

2.4.3 Discussion – FOSS component substitution at the subsystem level

Information needs:

The information needs required to perform component substitution at the subsystem level focus on three major aspects. There is a need for the involvement of user/domain experts as an integrated information source for the recovery/analysis process. There is also a need for the availability of higher level dynamic information, to provide

the basis for a behavioral analysis and the availability of some static structural information to supplement both user knowledge and behavioral information. Common to all of these information needs is that the level of granularity is coarse and their focus is mainly on the inter-relationships among major components of a system architecture.

Techniques:

The techniques review in this section focus on the following categories.

- Architectural recovery and reconstruction techniques, which are mainly interactive techniques that utilize mostly lower level reverse engineering techniques.
- Architectural transformation, to improve the quality and maintainability of an existing architecture.
- Metrics and dependency analysis techniques that provide a general guideline of the complexity and comprehensibility of the architecture and therefore, the ease a subsystem substitution might be performed.
- Visualization – providing high level views of the static program model and different views (the 4+1 views).

Information provided and applicability to component substitution:

At this subsystem level, the component substitution becomes more and more an instance of a hypothesis driven substitution process, involving a large number of information resources and techniques and providing different types of information. The goal at this level is to enrich the information provided by the different techniques, most of them were covered in the previous sections of this report, with both, more domain expertise and higher-level information. At this level of granularity, techniques become secondary, and the user/domain experts' expertise becomes one of the major resource to support the substitution. All of the techniques covered in this section focus on delivering high level views, beacons that can be used to refine existing hypotheses about the architecture and the components to be substituted, as well as their dependencies on the remaining system.

Table 18 Overview FOSS analysis at the subsystem/architectural level surveyed

Technique	Application
Architecture extraction	Comprehension, dependency analysis
Architecture Reconstruction	Validating of performed substitution
Architectural Transformation	Adaptation of architectures, legacy systems conversion
Architectural metrics	Quality evaluation, evaluation of maintainability, comprehension
Visualization techniques	4+1 view, comprehension, impact analysis
Impact analysis, tracing	Dependency analysis

2.5 Documentation

It is widely accepted that documentation can significantly improve a maintainer's ability to understand a system and make efficient changes to it. Despite this, many developers are still reluctant to use anything but the source code for maintenance tasks, as documentation is generally considered abstract and not up-to-date. However, many agree that the use of documentation is more significant if the information it contains is traceable between software models. Furthermore, documentation contains information, more specifically domain specific, functional and non-functional requirements that are otherwise not either reflected or easily found in the source code.

Table 19 Analysis techniques related to documentation and domain knowledge

Technique	Information need (input)	Information provided	Application	Static	Dynamic
Domain knowledge retrieval	Domain expert, documentation, source code queries, domain model	Functional/non-functional requirements	Comprehension, architectural recovery, architecture analysis	Y	N
Data mining, knowledge discovery	Data, traces, data bases, logs, domain expert	Patterns, prediction, clustering	Discovery of knowledge that is not directly identifiable in the source code	Y	Y
Traceability	Domain model, program model, documentation, domain expert	Traceability from requirements (functional to source code	Maintenance, comprehension, verification	Y	N

2.5.1 Domain knowledge

Software programs are solutions to problems from some application domains. One of the major goals of program comprehension is thus to understand the domain semantics of source code, i.e. to understand the functionality of the source code in terms of problem domain. Regardless of comments in source code, domain knowledge is expressed by 1) syntactic information of programming language and 2) identifier names. For example, the following assignment statement written in Java:

duration = eTime – sTime;

is to compute a *duration* by subtracting *start time* from *end time*, in which "duration", "eTime", and "sTime" represents domain concepts/objects, and "=" and "–" describes their relations. An experienced programmer may easily understand the above statement. However, automatically discovering domain semantics in source code is very difficult. As an example, above statement may also be written in the following form:
d = e – s;

Comparing these two, the former expresses computational intent in human oriented concepts that exist in the context of problem domain, while the later uses restricted identifier names and does not reference the human oriented context of domain knowledge [BIG94]. Although these two statements are identical to compilers, they have significant differences in terms of domain semantics discovery and program comprehension.
In this chapter, we introduce several software comprehension approaches that aim to facilitate software comprehension by recovering domain semantics in source code.

2.5.1.1 Program Plan

Traditionally, recognizing domain concepts in source code has been addressed by *program plan* research [LS86, RIC90]. A program plan is a pattern of syntactical structure of programming language that describes a particular computational intent. For example, the following code fragment

```
x = s[t];
t = t - 1;
```


can be identified as a program plan that *pops an element of a stack*. To recognize program plans, program comprehension tools usually adopt a language parser to represent source code in abstract syntax tree or graph. A program plan thus can be defined by a pattern in the tree/graph. Most of earlier systems [LS86, RM90] are based on graph parsing. Recent tools [MNS95, AFC98, and Ke01] can be also classified into this category, which are trying to identify concepts in the software design domain, such as design pattern or software architecture.

2.5.1.2 Concept Assignment

Biggerstaff et al [BIG94] observed that when a programmer starts to build an understanding of an unknown program, he/she must create or reconstruct the "informal, human oriented" expression of computational intent through a process of "analysis, experimentation, guessing and crossword puzzle-like assembly". As the domain concepts are discovered and interrelated, these concepts are simultaneously associated with or assigned to the specific implementation details, which can be considered as the concrete instances of those concepts. The problem of discovering these domain concepts and assigning them to their implementation instances within a program is the *concept assignment problem* [BIG94]. One of the simplest operational models for the concept recognition in source code is to view it as a program parsing model, such as program plan recognition [RIC90]. Biggerstaff et al [BIG94] argued that software model based on programming language parser are necessary but insufficient for the general concept assignment problem because the signatures of domain concepts are not constrained in the ways that are convenient for parsing technologies. For example, a Java language parser may recognizes "=" and "-" in the statement, but it is not possible to recognize these human oriented concepts, such as time and duration.

2.5.1.3 Domain Model and Inference

The use of informal knowledge such as identifier names and comments, as well as more formal analysis approaches such as parsers, can be used to recognize domain concepts. Two techniques used by Biggerstaff's [BIG94] approach are: 1) naïve assistant facilities, and 2) intelligent assistant facilities. The naïve assistant facilities assume that the user is the intelligent agent and the naïve facilities provide simple but computationally intensive services to support that intelligence. The intelligent assistant facility, which is called DM-TAO (Domain Model – The Adaptive Observer), is more experimental and attempts to provide a limited amount of intelligent assistance in performing concept assignment. These two techniques are briefly demonstrated in [BIG94], based on several scenarios to demonstrate how they can be applied to simplify and accelerate the concept reorganization. In the scenario where the user is the intelligent agent, he/she may follow the suggestive variable names, function names, or syntax patterns as the clues, and use naïve assistant facilities, including source code browser, call graph viewer, program slicer, etc. They are used to confirm or refute their guess until certain threshold of confidence is reached. The use of naïve assistant facilities requires a wide variety of viewing, analysis and query tools, which provide the mechanism for creating "opportunistic associations and juxtaposition of information" [BIG94].

A more promising approach for concept assignment is the use of clues provided by intelligent assistant facilities, i.e. to scan the code and present a list of candidate concepts based on the domain model DM-TAO. The results are used to create a rough sense of the conceptual highlights of the code being studied or to serve as the starting point for further investigation using the naïve assistant facilities. DM-TAO can answer several kinds of questions about source code: 1) Conceptual highlights: look for any concepts that correspond to some concepts in the domain model; 2) Conceptual grep: look for instances of a user-specified concept; and 3) What' this: propose a concept assignment for the arbitrary selected code. In order to accomplish the concept assignment, DM-TAO uses a domain model (DM) to drive a connectionist-based inference engine (TAO), similar to [FFGL88]. The DM is built as a neural network, in which each concept is represented as a node and relationships between nodes are represented as explicit named links. The information associated with each concept includes: the typical features that characterize the concept, its relationships to other concepts in the domain, relevant informal knowledge – such as the terminology likely to be used by a programmer when referring to this concept in code, the syntactic and/or conceptual context this concept is likely to occur in, etc. [BIG94].

In the inference engine – TAO, the nodes serve as the processing units of the network and generate appropriate signal strengths or activation levels as a nonlinear function of the input. The signals are propagated throughout the network via a controlled spreading activation process, which continues until the concept nodes compute their activation levels. If the computed output of a concept node is higher than a certain value, called the recognition threshold, then the domain concept node is predicted to be present in the corresponding section of code from which the relevant clues were extracted [BIG94]. The DM-TAO also can be adjusted through a training process, which is effected in two stages: 1) the network is initialized with *a priori* knowledge from the domain model regarding the degree of the association between two connected concepts (a qualitative assessment of low, medium or high

provided by the domain builder); 2) the network weights are adjusted in a performance driven manner using qualitative relevance feedback from users regarding the validity of the tentative concept assignments.

2.5.1.4 Discussion and review

Biggerstaff et al [BIG94] conducted several experiments to evaluate the DM-TAO inference engine used in the DISIRE system. In one experiment, they chose three files (600 lines of C code) containing data definitions within a problem domain. First, a manual analysis was performed on these files to identify the most important concepts to understand the data. There were 27 domain concepts in this set, of which only 20 were defined in the domain model (DM). Next they asked DM-TAO two kinds of questions: 1) What are the important domain specific concepts in these files, and 2) For a specific code segment, what is the concept most closely associated with it? The approach based on DM-TAO recognized all the concepts defined in the DM and produced 3 false positives, which were attributed to the fact that the system was only weakly trained.

LaSSIE System

The LaSSIE [DEV91] is a domain specific, model driven, and Description Logics based question answering system that can help software maintainers discovering domain specific information in software implementation (Figure 25. The user of LaSSIE describes a **p**rivate **b**ranch e**x**change (PBX) operation in a telephony network, and LaSSIE will find an instance of this operation in the source code.

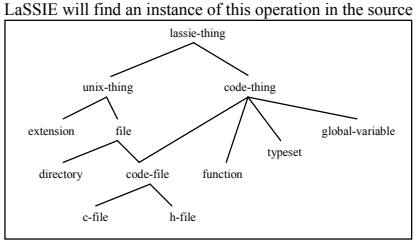


Figure 25 Code Model of LaSSIE [BAA03]

The inference, browsing, and classification services of LaSSIE provide additional flexibility in formulating queries. For example, the LaSSIE user can submit a query – "*How can I disconnect a trunk from a call because of a stimulus from a user*?" in natural language, and LaSSIE performs various kinds of taxonomic inference (e.g. relating drop to disconnect) to retrieve an answer that is semantically what the user wanted, but is syntactically very different.

Knowledge Base and Reasoning

LaSSIE contains two Description Logics knowledge bases: a *domain* model and a *code* model. The code model is implemented with a simple ontology of source code elements as the following Figure 30shows, which is derived empirically from the needs of software maintainers. The knowledge-base is populated automatically from the source code. The domain model is reverse engineered from the source code by domain experts. It contained knowledge about the PBX telephony domain. Part of the ontology is shown in the following figure (Figure 26).

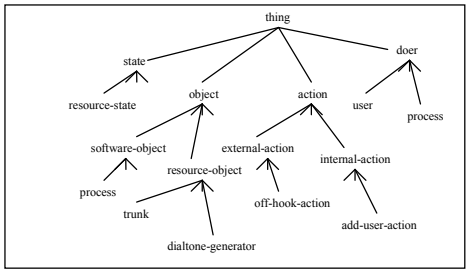


Figure 26 Domain Model of LaSSIE [BAA03]

The code model is able to solve many query problems by identifying major concepts in source code, such as variables, functions, files, etc. For example, to answer the question "what is the data type of the variable *error-value*", such question can be answered straightforwardly without any semantic noise such as *compute-error-value*, or *error-value-lookup-table*, etc. In the case where problem domain information is the desired result of a search, the domain model plays a role. For example, a maintainer may want to know "what kinds of actions a user of the system can take by themselves?" Such question may be simply answered by a Description Logics query:

$$Action \bigcap \forall initiatedBy.User$$

and LaSSIE system will find all the concepts subsumed by that expression.

The LaSSIE system also contains support for defining new domain concepts identified by maintainers during discovery. For example, the above query can be defined as a new concept

$$User\_Action \equiv Action \bigcap \forall initiatedBy.User$$

One observation using the LaSSIE system is that most domain queries are followed by code queries. For example, after exploring the domain model to discover the significance of a "connect action", the maintainer will typically ask, "What are the functions that implement it?" The integration between the two models can answer such questions. For examples, all function that implement connect actions will be:

$$Function \bigcap ConnectAction$$

and variables used in functions that implement user actions will be:

$$Variable \bigcap \forall usedInFunction.UserAction$$

2.5.2 Data mining and knowledge discovery

At the core of the Knowledge discovery and data Mining (KDD process are the data mining methods for extracting patterns from data. These methods can have different goals that depend on the intended outcome of the overall KDD process. It should also be noted that several methods with different goals may be applied successively to achieve a desired result. For example, to determine which customers are likely to buy a new product, a business analyst might need to first use clustering to segment the customer database, then apply regression to predict buying behavior for each cluster. Most data mining goals fall under the following categories:

- *Data Processing:* Depending on the goals and requirements of the KDD process, analysts may select, filter, aggregate, sample, clean and/or transform data. Automating some of the most typical data processing tasks and integrating them seamlessly into the overall process may eliminate or at least greatly reduce the need for programming specialized routines and for data export/import, thus improving the analyst's productivity.

- *Prediction:* Given a data item and a predictive model, predict the value for a specific attribute of the data item. For example, given a predictive model of credit card transactions, predict the likelihood that a specific transaction is fraudulent. Prediction may also be used to validate a discovered hypothesis.
- *Regression:* Given a set of data items, regression is the analysis of the dependency of some attribute values upon the values of other attributes in the same item, and the automatic production of a model that can predict these attribute values for new records. For example, given a data set of credit card transactions, build a model that can predict the likelihood of fraudulence for new transactions.
- *Classification:* Given a set of predefined categorical classes, determine to which of these classes a specific data item belongs. For example, given classes of patients that corresponds to medical treatment responses; identify the form of treatment to which a new patient is most likely to respond.
- *Clustering:* Given a set of data items, partition this set into a set of classes such that items with similar characteristics are grouped together. Clustering is best used for finding groups of items that are similar. For example, given a data set of customers, identify subgroups of customers that have a similar buying behavior.
- *Link Analysis (Associations):* Given a set of data items, identify relationships between attributes and items such as the presence of one pattern implies the presence of another pattern. These relations may be associations between attributes within the same data item (*'Out of the shoppers who bought milk, 64% also purchased bread'*) or associations between different data items (*'Every time a certain stock drops 5%, a certain other stock raises 13% between 2 and 6 weeks later'*). The investigation of relationships between items over a period of time is also often referred to as 'sequential pattern analysis'.
- *Model Visualization:* Visualization plays an important role in making the discovered knowledge understandable and interpretable by humans. Besides, the human eye-brain system itself still remains the best pattern-recognition device known. Visualization techniques may range from simple scatter plots and histogram plots over parallel coordinates to 3D movies.
- *Exploratory Data Analysis (EDA):* Exploratory data analysis (EDA) is the interactive exploration of a data set without heavy dependence on preconceived assumptions and models, thus attempting to identify interesting patterns. Graphic representations of the data are used very often to exploit the power of the eye and human intuition. While there are dozens of software packets available that were developed exclusively to support data exploration, it might also be desirable to integrate these approaches into an overall KDD environment.

From the above discussion one can see that data mining is not a single technique. Any method that will help to get more information out of data is useful. Different methods serve different purposes, each method offering its own advantages and disadvantages.

2.5.3 Traceability [IBR05]

Tryggeseth [TIG97] had demonstrated in his experiment that documentation significantly improves maintainer's ability to understand a system and make efficient change to it. Despite this, many developers are still reluctant to use anything but the source code for maintenance tasks, as documentation is generally considered abstract and not up-to-date. However, many agree that the use of documentation is more significant if the information it contains is traceable between software models. Software models are the work products such as code, design and requirements that differ from one another in terms of granularity. Code is a software model that emphasizes on the detailed granularity, while requirements are another set of software model that describe a system from the user's point of view.

There is a need to relate from one model to another e.g. from a requirement to its implementation code or design to test cases. This is called traceability. Ramesh [RAM01] relates traceability as the ability to trace the dependent items within a model and the ability to trace the corresponding items in other models. Pursuant to this, Turner and Munro [TUR94] assume that software traceability implies that all models of the system are consistently updated. Tracing such kind of traceability is called *requirements traceability* [RAM01]. Many software engineering standards governing the development of systems (e.g. IEEE/EIA 12207) emphasize the need of requirements traceability to be included into documentation and treat it as a quality factor towards achieving process improvement. The issue is that not much elaboration on what types of information is needed and how a strategy to achieve this are described in the development standards and guidelines. The developers normally prepare some traceability relationships between software components at higher abstraction levels and take least effort to integrate both the high level and the low level software models e.g. a requirement to its implementation code or vice versa. The ability to implement such requirement traceability would allow a maintainer or management to visualize the

impacts prior to actual change. This will greatly help in taking appropriate actions with respect to decision making, schedule plans, cost and resource estimates.

Static relationships are software traces between components resulting from a study of static analysis on the source code and other related models. Dynamic analysis on the other hand, results from the execution of software to find traces such as executing test cases to find the impacted code. Traceability provides a platform for impact analysis. One can classify three techniques of traceability.

1. Traceability via *explicit links. Explicit links* provide a technical means for explicit traceability e.g. traceability associated with the basic inter-class relationships in a class diagram modeled using UML.
2. Traceability via *name tracing. Name tracing* assumes a consistent naming strategy and is used when building models. It is performed by searching items with names similar to the ones in the starting model.
3. Traceability via *domain knowledge* and *concept location. Domain knowledge* and *concept location* are normally used by experienced software developer tracing concepts using his knowledge about how different items are interrelated.

2.5.4 Discussion – Documentation, domain modeling and traceability

Information needs:

Recovery domain knowledge from existing systems is an inherently difficult and challenging task. The challenge lies in both extracting and inferring knowledge from the available sources. Information sources for the reviewed techniques include domain experts and their knowledge, source code (program models) and databases (both their structure and the information stored within).

Techniques:

The techniques review in this section focus on the following categories.

- Domain modeling. Selecting an appropriate model to store any extracted domain knowledge plays an important role for further processing of the information. Depending on the underlying model, e.g. ontology based, relational or any other form, the ability to query and infer domain knowledge will be affected.
- Domain knowledge recovery. There exist two major approaches for domain knowledge discovery. Extraction of knowledge from the domain expert (human) and storing this information for further processing. Secondly, the ability to infer domain knowledge by analyzing and reasoning about the source code. Techniques not covered in this survey include natural language processing to analyze existing documentation to derive domain and application knowledge
- Traceability. Probably one of the most challenging and important issues of software engineering. The ability to trace requirements (functional/non-functional) to the source code and visa versa.

Information provided and applicability to component substitution:

From a documentation and traceability point of view, the challenge is to create traceability between existing source code and the available documentation. The challenge is manifold, due to the fact the documentation might be incomplete, inconsistent with the source code or might not exist at all. However, recovering some of the domain and application expertise that influenced the original design and implementation of a system is inherently difficult.

3 Tool Survey

There exist several surveys on program comprehension, reverse engineering and software visualization tools [HAM01, CHU05, TIP93]. In this survey, the focus is on tools that are not covered by these existing surveys. The previous section of this report covered techniques, their information needs and the information they provide to support component substitution at various levels of abstraction. In what follows, an initial survey of tools that implement and utilize some of these techniques is presented. From a tool perspective, the classification differs from the previous chapter, due to the fact that many of these tools are not focusing solely on one abstraction level, but often integrate different techniques to support the component substitution process at various levels of abstraction. The techniques utilized by these tools depend on the type of component supported (FOSS or COTS), the level of abstraction and the component substitution process supported by these tools.

3.1 COTS

3.1.1 COTS dependency analysis tools

There are many tools and techniques that provide visibility into COTS components. The choice of tool or technique is based on the type of component and the type of substitution to be performed. For a particular component, one of the most useful mechanisms are those targeted towards peering through the component's outer boundary. There exist also techniques that can assess a group of components working in unison. All of these mechanisms represent "observation posts" that provide insight into off-the-shelf components.

Intra-component visibility: Tools that observe the behavior of an individual component. These tools seek to identify a component developer's assumptions with respect to the intended use of the component. These tools provide in general static or dynamic visibility into parent/child relationships, thread performance, resource utilization, signal and event disposition, system calls (including parameters and return codes), user stack, and open files. Examples for such tools are crash under UNIX or ProcessViewer under Windows XP

Inter-component visibility: Tools that observe the behavior of two or more components. They seek to understand the relationship between components, using the inter-component visibility among components. Techniques used include both static and dynamic visibility analysis of logical and physical protocol streams, state information, procedure calls, and data exchange. Such snooping is applicable to software component interfaces and hardware (physical) interfaces. Snooping can occur at any point where data and control are extended past the boundary of a component's environment. This can include inter-component communication across processes (and also processor), boundaries through parent/child communication, remote procedure calls, client/server communication, and dynamic data exchange. Likewise, such snooping is not limited to network traffic (e.g., Ethernet, FDDI, etc.), but can be used on other physical transport layers such as RS-232 or SCSI. An example of such a tool would be etherfind (or snoop) under Unix or DDESpy under Windows XP.

3.1.2 COTS profiling and tracing tools for Java

The *Java Object Instrumentation Environment* (JOIE) [JOE06] is a framework for safe Java bytecode transformation, developed at Duke University. It provides both low-level and high-level functionality to extend or adapt compiled Java classes. The low-level interface allows manipulation of the bytecode itself, whereas the high-level interface provides methods for inserting new interfaces, fields, methods or whole code slices. JOIE extends Java class loaders with load time transformation. The JOIE class loader allows the installation of bytecode transformers. A transformer can insert or remove bytecode instructions and alter the class file being loaded.

The *Bytecode Instrumentation Tool* (BIT) [LEE96], developed at the University of Washington, is a collection of Java classes that allows users to insert instructions to analysis methods anywhere in the bytecode, so that information can be extracted from the user program while it is being executed. BIT was successfully used to rearrange procedures and to reorganize data stored in Java class files. An application, called *ProfBuilder* [Coo98], was built on BIT, and it allowed for rapid construction of different types of Java profilers.

The Bytecode Engineering Library (BCEL) [BCL06] developed by the Apache Software Foundation is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to implement

desired features on a high level of abstraction without handling all the internal details of the Java class file format. Unlike other bytecode manipulating tools, the BCEL is intended to be a general purpose tool for bytecode engineering. It gives full control to the developer on a high level of abstraction, and it is not restricted to any particular application area [BCL06]. BCEL is already being used successfully in several projects such as compilers, optimizers, obfuscators, code generators and analysis tools.

The Java programming assistant (Javassist) [JAV06] is a reflection-based toolkit for developing Java bytecode translators. It is a powerful class library for transforming Java bytecode, and it enables Java programmers to modify a class file before the JVM loads it, and to define a new class at runtime. The main feature of this bytecode operating library is that it allows users to access Java bytecode at the high source code level, instead of in the low bytecode instruction level. Unlike other similar tools, programmers can modify a class file with source-level vocabulary. Users do not have to have detailed knowledge of the Java bytecode and the internal structure of class file. Javassist can compile a fragment of source text on line, for example, just a single statement, and then inserts it into the Java bytecode. This ease of use is a unique feature of Javassist compared to other similar tools.

3.2 FOSS

3.2.1 FOSS Instrumentation and profiling tools

The following section discusses some existing profiling and instrumentation tools available either as open source or commercial tools. The relevance of these tools for component substitution is to provide (1) traces for further dynamic analysis, (2) Coverage tools can also be used to evaluate the quality of profiles and test cases used to create traces.

"*Clover*" is a commercial code coverage analysis tool, developed by Cenqua Pty Ltd. [CLO06]. It is used to discover the sections of code that are not being adequately exercised by unit tests, and then used to measure testing completeness. *Cover* utilizes source code instrumentation. It copies and instruments a set of Java source files. The output instrumented java source will then be compiled by a standard Java compiler. *Clover* measures three types of coverage analysis: statement, branch and method coverage. Moreover, *Clover* provides fully integrated plug-ins for many popular integrated development environments, such as *Eclipse*, *NetBeans*, *JBuilder* and *JDeveloper,* and works seamlessly with *JUnit*.

"Daikon" is a dynamic invariant detector, developed by the Program Analysis Group at the Massachusetts Institute of Technology [DAI06]. An invariant is a property that holds at a certain point or points in a program. Invariants are often seen in assert statements, documentation, and formal specifications. *Daikon* dynamic invariant detector runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions. *Daikon* can detect properties in Java, Perl, C, C++, and IOA programs, in spreadsheet files, and in other data sources. *Daikon* provides a plug-in for Eclipse which instruments Java source code, obtains trace information, analyzes those traces, and creates appropriately annotated Java source code to represent the invariants found by *Daikon* while running the instrumented program.

"*query and instr*" are Java test coverage and instrumentation toolkits developed by Glen McCluskey & Associates LLC. They are used for parsing, querying, and instrumenting Java source programs, and are ideally suited for applications such as test coverage, metrics, instrumentation, extraction of information, documentation tools, program tracing, and so on. The toolkits have two packages: *query* and *instr*. The *query* package is used to parse Java source programs into an internal tree form. It also contains classes and methods for querying the parse tree. The tree may be annotated, i.e., it may have information added to it which could be dumped out at a later time. This is the basis for program instrumentation. The *instr* is a test-coverage and instrumentation package, which is built on the *query* package. It supports method and statement source instrumentation, and test coverage. These can be used as actual end-user programs to instrument code, or as the basis for customized applications.

3.2.2 Source code query tools

Source code query techniques are some of the most successful and most widely used approaches to guide programmers during program comprehension and component substitution. The success of these techniques is based on the availability of both relatively powerful and easy to use tools that support these techniques. There exists a number of source query tools that support the browsing, exploration and querying of underlying program models at various levels. The tool survey focuses on two categories: lexical and syntactical queries, which are covered in more detail in the next sections.

3.2.2.1 Lexical Queries on Source Code

The success of *grep* is evident with the family of tools that it has spawned. One of grep's limitations is that matches must appear on a single line.

Other limitations:

(1) *grep* may return a large number of matches that make it difficult to find the most relevant one. The strings matching the search target may be difficult to find (because they are in the form of a line) and then results themselves may also need to be searched. In addition, since each match consists of a single line, it might not provide enough contexts to interpret the statements returned.

(2) No approximate searches are supported , matches can not be approximated and must be exact according to regular expression rules. If there is a spelling mistake in a search target, there is no facility in *grep* to deal with that.

(3) No semantic searches – all input to these lexical searches are treated as text. When searching source code, there is no way to limit the search domain, e.g. identifiers or comments.

4) No memory – Search targets or contexts are not stored and cannot be revisited for refinement or modification.

5) No browsing – The search results cannot be browsed like hypertext – clicking on a result to display the original place in the source code.

The *cgrep* (context grep) [CC96] tool addresses this issue by treating the input as a character stream and interpreting the new-line character as ordinary text, so that it can return matches with arbitrary sizes.
The *agrep* tool [WU_92] is another extension to *grep* with three modifications. It allows approximate matches by permitting a user-specified number of substitutions, insertions, or deletions, which could be used in situations when the maintainer didn't know the extract name of an identifier. Second, instead of returning a single line, agrep can return matching records, such as entire email messages. Finally, it allows the logical combination of patterns using AND or OR.

LSME is similar to awk [AHO79], in the sense that LSME searching specifications are closer to a script or a program, other than a command-line utility. For example, a maintainer who wants to query function calls between functions from the C source code for gcc. One of the problems with the LSME tools is its accuracy. For example, LSME tools may generate all the function calls in the source code, and some information are false positive [MUR96]. However, Murphy considers the flexibility a maintainer gains in such lightweight lexical tools is a fair trade for accuracy.

Discussion on the Limitations of Lexical Query Tools

Lexical query tools are successful in the way that they are flexible and tolerant. These tools are programming language independent, and thus are not sensitive to the condition of the source code. They are capable of scanning various kinds of software artefacts such as data files or documentation files. Besides, in the situation when the system is not compilable (e.g. it is in the midst of development), such tools is also applicable.
The major limitations of lexical query approaches are

1. The "semantic noise" [WEL97] in the query results, i.e. the false positive. Typical semantic noise includes a) identifiers with longer names that include the specified keyword; b) comments that include the specified keyword; and c) identifiers and comments that match the specified regular expression but are not desired.
2. Source code is more structurally complex than what can be captured by lexical approaches. Paul and Prakash have discussed many limitations of regular expression matching [PAU94].
3. Writing a pattern to distinguish the nesting structure of programming language statements is difficult, and sometimes is even impossible. Writing a pattern to distinguish variable declarations, characterized by order-independent strings of attributes (e.g. extern long int x) is difficult and unwieldy.

3.2.2.2 Syntactical Queries on Source Code

A number of approaches (CIA [CHE90], CIA++ [GRA92], grok [HOL96], RPA (Relation Partition Algebra) [FEI98], sgrep [BUL02], etc) have been proposed using relational model to represent source code structures. Users of CIA system can query the information using the relational query language provided by the database system.

The conceptual model for a C program defines the objects and relationships at a selected level of abstraction. It serves as a requirements specification for the information abstractor and determines the extent of knowledge available in the database [CHE90], shown in Figure 27 In this model, five types of objects are defined to describe

the C language – *File, Macro, Data type, Global variable and Function*. Each object has a set of attributes. For example, a function has attributes such as 1) file that this function is defined in, 2) data type the function returns, 3) name of the function, 4) whether it is a static function, 5) its start line number and 6) its end line number. The conceptual model does not explicitly define the order of the attributes and their storage format, but leaves the details to the relational schema.

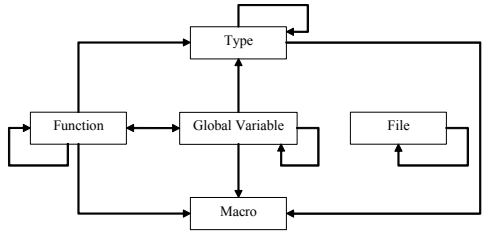


Figure 27 Conceptual Model of CIA [CHE90]

A relation - *reference* is defined in the conceptual model. If an object A has a reference relationship with object B, then A cannot be compiled and executed without the definition of B. The following table shows all the meaningful reference relationships among the five kinds of object in C program.

Table 21 Reference relationships among C programs

Object kinds #1	Object kinds #1	Interpretation
File	File	File1 includes Files2
Function	Function	Function1 refers to Ffunction2
Function	Global Variable	Function refers to Global Variable
Function	Macro	Function refers to Macro
Function	Type	Function refers to Type
Global Variable	Function	Global Variable refers to Function
Global Variable	Global Variable	Global Variable1 refers to Global Variable2
Global Variable	Macro	Global Variable refers to Macro
Global Variable	Type	Global Variable refers to Type
Type	Type	Type1 refers to Type2
Type	Macro	Type refers to Macro

A C language parser extracts these defined objects and relationships from a program, and stores the information in an INGRES database. Users of CIA can query the information through InfoView - a collection of tools specifically designed to access the conceptual model, or through QUEL - the relational query language of INGRES. Three major types of information retrieval are illustrated – 1) info: retrieval of attribute information of an object; 2) rel: retrieval of relationships between two object domains; and 3) view: view the definition of an object.
CIA++ [GRA92], grok [HOL96], RPA (Relation Partition Algebra) [FEI98], sgrep [BUL02] etc are all based on relational models. They extend the CIA system by 1) supporting a variety of programming languages, 2) providing more fine-grained source code objects, and 3) providing more expressive query languages.

The query tool based on Source Code Algebra (SCA) introduced in [PAU94] is another approach to set up a formal framework to support source code queries. It is based on relational algebra – the foundation of relational database, cannot support a wide variety of atomic (e.g. integer, string) and composite data types (e.g. while-statement *has* condition and body, or statement-list is a *sequence* of statements) in source code.

More recently, several graph based approaches (GUPRO [LSW01], CLG [Ku00], etc) have been proposed to overcome the limitations of the relational model [PAU94, KLI03]. These approaches commonly represent source

code in graph structures (node as source code object and edge as relation), and use GReQL [KUL99] to query the graph. In GUPRO [LSW01], source code is parsed into graph structures. The conceptual model of the graph structure is illustrated in Figure 28 in which four types of node, (Module, Class, Method, and Attribute) capturing major objects in source code, are defined. Relations between objects, as well as their restrictions, are also specified.

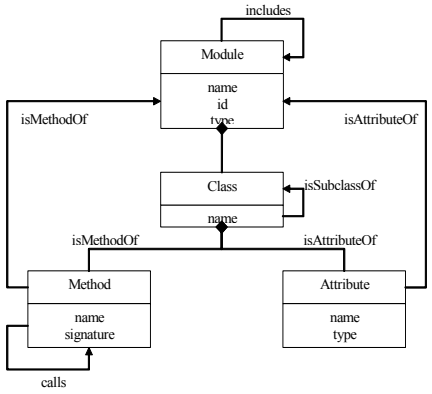


Figure 28 Conceptual Model of GUPRO [LSW01]

GUPRO accesses the graph structure using GReQL (Graph Repository Query Language) [KUL99]. GReQL is a pure declarative query language that provides a powerful means for analysis of entities and their relations in graph structure. A typical GReQL query consists of three clauses: FROM, WITH, and REPORT. The FROM clause declares graph objects (nodes and edges). The WITH clause specifies predicates that have to be satisfied by the declared graph objects. GReQL supports first-order predicate logic on finite sets. The path expression can be used to describe regular path structures, e.g. sequences, alternatives and iterations (reflexive and transitive closure) of paths in a given graph. The REPORT clause displays the query results in the form of table. GReQL queries can be nested in the REPORT part. [KUL99]

For example, the following query [LSW01] reports the name of each class and the name of the most general super class of that class:

```
FROM c, super : V{Class}
WITH c --> {isSubclassOf}* super
AND
outDegree {isSubclassOf} (super = 0
REPORT c.name, super.name
END
```

The FROM part declares two variables: c and super, of type Class. The WITH part restricts the possible assignments such that c has to be a subclass of super on arbitrary level and that super must not be a subclass of any other class. The REPORT part specifies to report the name of class c and the name of its highest super class.

Limitations of Syntactical Query Tools

In general, existing syntactical query approaches can be distinguished by their underlying source models and the query languages they provide. However, many of them share a common structure:

1. A repository that stores source code information according to a model;
2. Tools that populate the repository with structural and/or program flow information, such as language parsers or analyzers;

3. A query processor that handles queries by retrieving from the repository; and
4. An interface for the user to submit queries and obtain results.

Most of the existing approaches (e.g. CIA [GRA92], SCA [PAU94], grok [HOL96], RPA [FEI98], sgrep [BUL02]) are based on a relational model to represent source code structure have some inherited problems:

1. A relational model can not capture hierarchal relations between source code elements;
2. SQL-styled query languages lack the ability to express transitivity (with the exception of sgrep); and
3. Different tables have to be joined many times to traverse relational paths.

More recently graph based tools (e.g. GUPRO, CLG, etc) were introduced to overcome the previously discussed problem (2) and (3) of relational models. These graph based approaches support transitive closures and enabling the user to formulate regular path expressions. However, these approaches still cannot capture type hierarchies. Some other existing work uses network structures to represent source code. For example in Rigi, a special purpose semantic network data model is adopted to represent objects and relations in source code and using RCL language to manipulate them. The problem of these tools is the lack of query languages with well-defined semantics. The queries have to be procedural. While some of these tools are quite successful in achieving specific program comprehension tasks, they are restricted by their internal representations. These supported representations are typically neither intuitive nor flexible enough to help maintainers to construct an appropriate mental model. One possible reason is that the underlying models these tools use do not correspond closely enough to a programmer's own mental model or his/her expertise of a program. Therefore, programmers will have to match their own mental model with the models provided by the tools. This mismatch indicates a clear need for a unified model that bridges the conceptual gap between software representations and programmer's mental models.

Research on applying Description Logics in software engineering have been addressed in early works of the LaSSIE system and Welty's work. These approaches, however, are much more restrictive in their expressiveness of the adopted ontology languages and they lack optimized Description Logic systems. Research in cognitive science suggests that mental models may take many forms but the contents normally constitute an ontology [LAI81]. An ontology is a description of the concepts and relationships that can exist in a domain [BAA03]. In the software program comprehension domain, such ontology consists of various concepts from programming languages and techniques. It also includes a set of roles (relationships between concepts) characterizing the relations between entities in a program.

In order to understand a program, software maintainers spend considerable time in querying/searching source code. However, existing program comprehension tools supporting source code query have several shortcomings. They are, briefly:

- Lexical approaches (grep, awk, LSME [MUR96], etc.) can not capture the structural properties of source code, and the searching results are inaccurate and opportunistic.
- Syntactical approaches using relational models (CIA [CHE90], CIA++[GRA92], grok [HOL96], RPA [FEI98], GReQL [KUL99], sgrep [BUL02] etc.) are inadequate to represent the structures of source code.
- Existing approaches [BIG94, LI01, GOL02, MAR04, etc] that support discovery of domain information in source code need to be improved in terms of computational complexity and precision.
- Existing program comprehension tools based on Description Logics [DEV91, SEL94, WEL97, etc] manually maintain the relationship between domain ontology and source code ontology, which is time-consuming and error-prone.

3.2.3 Feature/Component level

The following section provides a review of tools covering techniques such as design pattern recovery, feature extraction and concept analysis tools. Common to all these tools is that they attempt to infer some type of domain knowledge/expertise from the source code, by identifying/extracting either design decisions in the form of design patterns or by attempting to recover functional requirements in the form of concepts and features from the source code.

3.2.3.1 Design pattern recovery

Several pattern-based reverse engineering tools have been introduced to facilitate software comprehension within the last few years. Most of these tools are based on pattern detecting techniques. However, only a very few of these tools have a solid theoretical basis. In what follows, a brief introduction and review of some of the tools and their underlying approaches is the system structure, and the results and experiences from applying these tools.

Pat

The Pat [KRA96] system is a reverse engineering tool that searches for design pattern instances in existing software. Within Pat, design information is "extracted directly from C++ header files and stored in a repository" [KRA96]. "The patterns are expressed as PROLOG rules and the design information is then used to search for all patterns" [KRA96]. Since the Pat system is designed to collect information just from C++ header files without semantic analysis of method body, the authors limit the considered patterns to five structural patterns introduced in the GoF book [GHJV94]: Adapter, Bridge, Composite, Decorator, and Proxy. The approach of the Pat system is simple: representing both patterns and designs in PROLOG and let PROLOG engine do the matching job. The recovering process starts from representing each pattern as static OMT diagram, and converting the diagram to PROLOG representation, which is one rule for each pattern. Another reverse engineering tools, ooCASE, is used to analyze C++ header files and store the information in the repository in OMT form, which will be translated into to PROLOG representation after. At last, a PROLOG query will detect all instances of design patterns from the two repositories. [KRA96]

In order to evaluate the quality of the design pattern recognition of the Pat system, the author used *precision* and *recall* [FB97], which is the most commonly used measure of retrieval effectiveness. Precision can be defined in this context as the ratio of the number of true patterns found by Pat over the number of all candidates found by Pat. Recall is the ratio of the number of true patterns found by Pat over the number of all true patterns existed in the software. The authors of the Pat system state that in the 4 benchmark applications, the precision ranges from 14 to 50 percent, and will be "much higher if Pat could also check for correct method call delegations" [KRA96].

AOL

In [AFC98], Antoniol et al present a conservative approach to recover design patterns from design and source code, which is mainly based on "a multi-stage reduction strategy using software metrics and structural properties to extract structural patterns for OO design or code" [AFC98]. The reason why the authors call this approach conservative is because any patterns in the code will be surely reported in the result. Similar to the Pat system, this approach also focuses on the same five structural patterns mentioned in [KRA96]. However, in addition to Pat, method delegation, which means a method delegates its responsibilities by calling another method of an associated class, is used as a design pattern constraints to reduce the reported false candidates. The process of this approach consists of four activities. The first is AOL (Abstract Object Language [Pe97], a general purpose design description language) representation extraction, which represents the C++ source code or design in AOL, and the second step is to analyze the AOL representation to generate an AOL abstract syntax tree. In the next step, relevant class metrics are extracted. Finally, pattern constraints like metrics constraints, structural constraints, and delegation constraint are applied to the AOL abstract syntax tree and the class metrics obtained in the previous steps and a set of pattern candidates are conducted. The authors claim that the constraints are all necessary conditions, and thus the output does not contain false negatives but it may contain false positives, which must be manually inspected by the user. [AFC98] By recovering design patterns from more than 14 codes of public and industrial systems, the authors provide a comprehensive experimental result analysis. In addition to measuring the results in terms of *recall* and *precision*, the reduction of candidates through the stage filters is also provided, with respect to Initial stage, metrics based filter stage, structural filter stage, and delegation filter stage. The authors conclude that, in public domain code case studies, the average precision is 55 %, and there is "an increase of about 35% using also the delegation constraint with respect to the use of structural constraints alone" [AFC98].

Columbus and Maisa

Columbus [FER01] is a versatile reverse engineering system that transforms C++ programs into a number of abstract representations, including UML class diagrams. Maisa [NEN00] is a metrics tool that analyzes the quality of a software architecture given as a set of UML diagrams. In [FER01], Ferenc et al present a technique for automatically recognizing design patterns from object-oriented (C++) source code by integrating Columbus and Maisa. The authors state that Columbus and Maisa pairs can be used to document and analyze software implemented in C++, and to verify the architectural design decisions during software implementation phase as well. The tools are applied by combing both Columbus and Maisa. The process used by these tools can be described in two major steps: The Columbus system first transforms the C++ source code into UML class diagrams. These diagrams are then traversed and matched against a set of predefined design patterns by Maisa, which consists of a set of CSP (Constraint Satisfaction Problem) formulas to describe each UML class diagram incorporated with additional information obtained from different type of UML diagrams. The authors are "studying the possibility of using dynamic information (such as sequence diagrams) for defining patterns more accurately"[FER01].

3.2.3.2 Dynamic analysis tools

Dynamic analysis of applications is used for program comprehension, dynamic optimization and application monitoring. Several tools and technologies have been developed. These tools, differ in the area of analysis and the type of instrumentation that they use. The following is a brief overview of these dynamic analysis tools [SAL06].

- Software Reconnaissance [WIL95] uses test cases to locate the source code that relates to a particular feature. In Software reconnaissance, a program is instrumented similar to the instrumentation used by typical test coverage tools. Afterwards, two different execution runs are started, the first set of program executions is using a particular program feature, the second set of execution runs does not executed the particular program feature. The difference among these executions runs is used to build a mapping between features and code.
- Concept analysis is used to identify the most feature-specific subprograms among all executed subprograms. A static analysis uses this subprogram to identify additional feature-specific subprograms along a dependency graph. While Reconnaissance provides support for well established analysis techniques, it is somewhat limited in the information detail it can provide.
- BEE++ [BRU93] is a C++ based object-oriented framework for the dynamic analysis of distributed systems. BEE++ considers the execution of a distributed system as a stream of events. Hence, BEE++ allows the customization of events and event views. Event and event view customization rely on the inheritance mechanisms of C++. The main drawback of BEE++ is that the instrumentation of the application has to be performed by the programmer manually.
- ATOM [AMI04] is a framework for the construction of program analysis tools. It takes a set of object files, a file containing the instrumentation routines, and a file containing the analysis routines. Instrumentation is started by invoking the Instrument procedure written by the user. This procedure allows the selection of the routines contained in the object code. Similar to AOP techniques it is possible to insert calls before and after certain points within the object code. Within ATOM, these points can be procedures or even assembly instructions. ATOM, however, has some drawbacks concerning program comprehension. First, the structure of the source code is no longer visible within the instrumentation routine since it works on the object file level. Second, inserting instrumentation code for different procedures starts to become difficult if the parameters stored within registers shall be considered.
- Form [SOU01] can be used to construct tools for analyzing the runtime behavior of standalone and distributed software systems. Form collects data during an application run and visualizes call graphs with UML sequence diagrams. Form avoids the instrumentation of the applications and uses the Java Virtual Machine Profiling Interface (JVMPI) to collect data. Since Form uses the Java Virtual Machine's profiling interface, it can analyze programs where the source code is unavailable. A disadvantage of this approach, however, is that the profiling interface is tuned for the performance analysis and hence, few notifications such as the invocation of methods or creation and termination of threads can be used for program analysis.

3.2.3.3 Concept analysis and Feature extraction [KNO05]

Existing tools and methodologies for architecture recovery and feature location have been used to extract and determine assets (e.g. components) In particular for architecture recovery, a number of tools have been developed that can be used to extract higher level views on the implementation of software systems. Tools are, for example Bookshelf [FIN97], Dali [KAZ99], or Rigi [MUL98]. They follow the Extract-Abstract-View Metaphor described in [22]. Most of these tools differ in the underlying fact extraction technique, in the methods and details of fact representation, and in the analysis and visualization techniques. In [EBE02], Ebert et al. introduced GUPRO which is an integrated workbench that supports program understanding of heterogeneous systems on arbitrary levels of granularity.

The SAR method described by Krikhaar [KRI99] concentrates on creating higher-level views on the architecture. The approach is based on Relational Partition Algebra and defines a process for selecting the information sources from which higher-level views are abstracted.

Riva proposed a view-based architecture reconstruction approach named NIMETA [RIV04] Similar to Krikhaar, the approach is based on relational algebra. NIMETA emphasizes the scrupulous selection of architectural concepts and architecturally significant views that are reflecting the stakeholders' interests.

Regarding feature location, a number of approaches exist. Concerning the feature location, in source code Wilde et al. presented pioneering work. They introduced the Software Reconnaissance approach that based on the execution of test cases determines features [WIL01].

Eisenbarth et al. based their approach on the Software Reconnaissance technique and extended it by using the concept analysis technique for determining features [EIS01]. A similar approach has been also presented by Wong et al. They analyze execution slices of test cases to determine the source code units that implement a feature.

Several existing tools of Formal Concept Analysis have been developed mainly for research purposes and also as learning tool in computer science courses. Most tools serve as a context editor, a concept calculator and a concept lattice drawing tool. Current projects of Formal Concept Analysis are listed below.

- *Toscana* [TOS01]

  Toscana is one of the components in the TOCKIT project [TOC01]. Toscana was formerly written in C++ and OWL, but suffered from typical legacy problems. As a result, the ToscanaJ project has been developed through the collaboration between DSTC, the University of Queensland and the Technical University of Darmstadt to recreate a Formal Concept Analysis tool called Toscana and to give the FCA community a platform to work with. The ToscanaJ suite comes with editors for different types of Conceptual Information System (with or without relational databases in the backend), but editing can be a complex task and this complexity is not exposed to the user of the final system (Figure 29). The principal FCA tools in this project are the following:

  ToscanaJ - a concept browsing front-end used as a concept viewer and browser with the following main features:

  - Display of simple and nested line diagrams
  - Color encodes the object contingent sizes (can be changed to extent). In addition, node size can be used for this type of information
  - The object set of interest can be filtered
  - Nodes in the diagram can be clicked to get highlighting as reading help
  - Diagrams can be exported as SVG, PNG and JPEG. Additional information about the way the diagram was achieved is exported as separate text files, via clipboard or within the SVG file
  - Different label contents, using data-specific SQL fragments
  - Additional database views can be opened from the diagram, e.g. a viewer using HTML templates where query elements get resolved
  - The database viewer interface is designed as a plug-in interface to make it very easy to extend ToscanaJ for other specific purposes
  - HTML descriptions can be attached to the schema, to diagrams and to attributes
  - The database viewers can be used for the attributes, e.g. to query an URL from the database which is then opened in an external browser

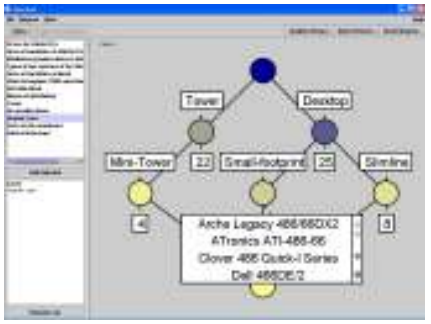


Figure 29: Screenshot of ToscanaJ [TOS01]

Another concept editor - Elba - focuses on conceptual schemas of relational databases. The following features of Elba allow users to create their own conceptual system of a system (Figure 30).

- Editing of diagrams and contexts
- Database wizard helps users connect to different databases (internal, JDBC, ODBC, MS Access files)
- Scale generators help create different types of diagrams
- The diagram layout and manipulation is based on n-dimensional structures, making editing diagrams easy while ensuring additive drawings
- Different manipulators can be used to override the attributes and to create arbitrary Hasse diagrams
- A grid can be used as a guide for neat layout. This is supported by all manipulators
- A very simple XML editor allows adding the descriptions of the different elements
- SQL scripts can be exported to port small databases from proprietary systems into using the embedded database engine
- An XML summary of the diagrams can be exported to analyze the realized diagrams, i.e. the coverage of the lattices by the objects in the database

Figure 30. The screenshot of Elba [TOS01]

Another concept editor tool called Siena [TOS01] offers similar features to Elba's without the need of a database. Users can edit conceptual schemas that store their data in memory (Figure 31).

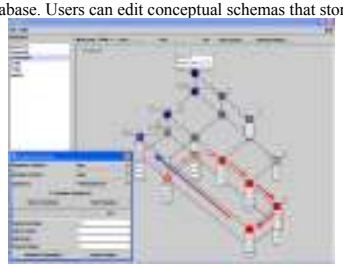


Figure 31. The screenshot of Siena [TOS01]

- Anaconda [ANA99]

Anaconda is an interactive editor for ConScript files which runs under Microsoft Windows. ConScript is the file format which is supported by *Anaconda f*or storing the data structures of Formal Concept Analysis (Figure 32). Anaconda provides in a Multiple Document Interface (MDI) a series of different editors for the different structures. The output can be prepared as well as manipulated independently. For example, a line diagram of the concept lattice of a formal context can be computed in Anaconda for a direct analysis on the screen or in a printout. However, the main purpose for the usage of Anaconda is to prepare input for Toscana [TOS01].

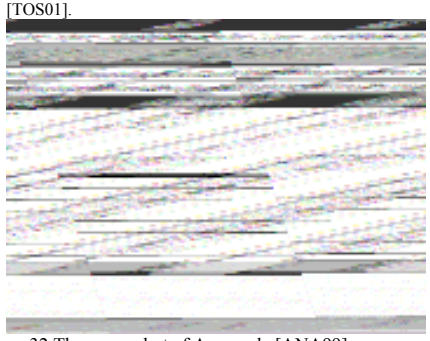


Figure 32 The screenshot of Anaconda [ANA99]

- DOS [DOS99]

Darmstadt Research Group focuses also on the use of Formal Concept Analysis in software Engineering. It has developed the three following concept-based programs:

- ComImp - a DOS program for contexts, concepts, concept lattices and implications
- Diagram - a DOS program for drawing line diagrams of concept lattices
- MBA - a DOS program for many-valued contexts

- ConExp [CON03]

ConExp or Concept Explorer, implements basic functionalities, needed for students and researchers in the field of Formal Concept Analysis. The main features of ConExp are the following.

- Context editing
- Automatic display of arrow relation
- Generation and display of concept lattice
- Editing of the concept lattice diagram
- Variable display of the diagram
- Highlighting of diagram nodes
- Calculation of Duquenne-Guigues base of implications
- Calculation of association rules

- Online Java Lattice Building Application for Concept Lattices [JAL04], [AUE04]

Lattices are graphically displayed as a Hasse-Diagram which is difficult to draw by hand. JaLaBA provides an online tool to create contexts and manipulate the resulting Concept Lattice. This application was created as part of the Ph.D. thesis of Maarten Janssen, called SIMuLLDA [JAL04]. JaLaBA first builds lists of the names of objects and attributes, and their relations. From this, it creates a list of FCA concepts. These are then put into a lattice drawing application to finish up the lattice, and exported to a printable format. A similar tool is provided by Auer [AUE04], called Formal Concept Calculator, which is a web-based tool that provides a wizard for creating contexts and visualizing concept lattices.

- Concepts [LIN02]

Lindig's Concepts tool that is distributed under the GNU Public License [GNU] as portable C source code. The Concepts tool is a command line application running under the UNIX environment. It was written by following Formal Concept Analysis theory to compute a lattice of concepts from a binary relation of objects and attributes. Concepts program's output could be presented in a plain text or in the Graphlet format [BRA99], a graph drawing tool written in C. This project has been extended by several researchers. It is mainly used as a mechanism to derive concept lattice graphs out of object and attribute lists.

Besides open source projects, there also exist commercial FCA tools/services, like the one provided by NaviCon [NAV04], a German company, offering commercial services and concept-based software, NaviCon Decision Suite, which is composed of Cernato and Toscana.

- The Cernato tool provides a complete overview of dependency, commonality and variability in the data. The user has full control of the analysis process at each moment and can individually examine and change at any time. Thus, the process retains transparency. This program provides data input functions and diagram viewer with clear structure. It also supports the unrestricted representation of complex data.

- TOSCANA is a front end Tool for graphical analysis and search data in SQL databases and serves for information system design. Toscana defines objects and characteristics in the data and computes their conceptual structure. These data are then converted into clear and readable diagrams and can also be graphically analyzed and examined by the tool.

Discussion of Formal Concept Analysis Tools

Current tools for Formal Concept Analysis provide the following main features:

- Context Editor: The function that allows user to input concept context conveniently and intuitively.
- Diagram Editor: The interactive tool for editing nodes and their properties. The edited data, in return, will be fed back to the context table.
- Diagram Layouter: The function that passes calculated concepts and their relationships to the graph drawing algorithm and visualize the concept lattice in hierarchical pattern.
- Concept Calculator: The function providing algorithm to calculate concepts and their relationships from the input context table.

- Output file format supported: The file format of the program's output.
- Input file format supported: The input file format that the program accepts.

On-line tools such as Formal Concept Calculator [AUE04] and JaLaBA [JAL04] are helpful for students and small research projects. They can serve as tools to create small examples to convey the idea of Formal Concept Analysis. However, the limitations of internet-based applications, such as poor response time, limited reliability and the inability to integrate these tools with others, limit their application for larger projects and data analysis.

Table 21: Summary table of Formal Concept Analysis Tools

Feature/ Tool	Context Editor	Diagram Editor	Diagram Layouter	Concept Calculator	Output Format Supported	Input File Format Supported
ToscanaJ		√	√		.svg, .png, .jpeg, .pdf, .eps, .emf	.csx (.xml)
Elba	√	√	√		.sql, xml	
Siena	√	√	√		.xml	
Anaconda	√	√		√	.csx	
DOS Suite	√		√	√		
ConExp	√	√	√	√	.cxt, .cex (xml)	.cex(xml),cxt, .csv, .oal
Formal Concept Calculator	√	√	√	√	.ps	
JaLaBA	√		√	√		
Concepts				√	script for Graphlet [BRA99]	.con (text file)
Cernato	√		√	√		.csv, .xml
Toscana			√			

Some of the limitations of the most commonly used tools are summarized below:

- ToscanaJ is a well-known FCA tool [TOS01], in particular due to its support for various graphic outputs and input file formats. It allows for a data exchange among FCA tools. The ToscanaJ developer team emphasizes the development of a general Formal Concept Analysis framework and promotes the input file format .csx, which is based on XML to be accepted as a standard exchange format in the Formal Concept Analysis research community. However, the input of ToscanaJ is still dependent upon its editor, Elba and Siena. The algorithm to layout the diagram is not an independent module to be easily extended by other researchers.
- Concepts by Lindig, C. [LIN02] is another common FCA tool that is used widely in the research community as a data analysis mechanism in FCA-based applications. It provides the algorithm to calculate concepts and its lattice. However, it is written in C and it is only available for the UNIX/LINUX operating system. The output in graph script format is also tied to the program Graphlet [BRA99] that also requires the UNIX operating system. The limited portability is one of the major restrictions of this tool.
- ConExp [CON03] supports most of the main features of FCA. It is assumed to be the easiest tool to perform FCA analysis since users can actually input the context file through simple text file providing a set

of objects and attributes. Therefore, this program is suitable for FCA studies as it is simple for the users who do not have a strong FCA background. Table 21 is the summary of features supported by each FCA tool.

In terms of research and development, most of the tools presented in Table 21 are not suitable for use in FCA application development. One reason is that they are not designed to support extensions by other newly developed programs or features. Some are designed to be learning tools so there are limitations with respect to their scalability. Some tools also require the input to be in a specific format which has to be created by a specific context editor. Some tools are not open-source program. Thus, they do not facilitate the connectivity to other add-on components. However, the Concepts program by Lindig is designed to be used in both FCA application and research. The fact that Concepts is also an open-source program enables it to be further developed. It is also possible to implement an automated FCA application by calling Concepts and transform the output of this program to other formats required by other graph layout programs.

3.2.4 Software Visualization Tools

A large number of architectural recovery tools and software visualization tools were reviewed as part of the previous state-of-the-art report on architectural recovery [RIL03]. The majority of software architecture visualization tools only support browsing [SIM99]. Tools such as Rigi [MUL98], PUNS [MUR96], Dali [KAZ98], and Software Bookshelf [FIN97] display software architectures and allow users to explore them, but have only primitive query mechanisms. Other tools allow the user to query and build views, such as CIA [GRA92], LSME [MUR96], and ManSART [YEH97]. However, these tools operate only on architectural level facts, they do not use the architecture as a means for organizing or accessing the information space underlying the abstraction.

For tools that automatically create visualizations of structural software models, syntactical restrictions to allow easy drawing by humans are irrelevant. Usually, such tools present software entities and their relations as graphs. Rigi [MUL98], an early and very influential tool for software structure visualization, still creates simple box-and-line diagrams. However, it provides extensive navigation facilities that allow the user to create views of different parts of the visualized system on different levels of abstraction. The more recent tools SHriMP [STO95] and Portable Bookshelf [FIN97] provide fisheye views of nested graphs. This technique presents at the same time the details and the context of a part of the visualized system. The use of the third dimension promises an increased information density and larger degree of freedom for graph layouts. After the introduction of three dimensional graph layouts to software structure visualization by Reiss [REI95], many different ways of using the third dimension were explored. The results of empirical studies that compare the effectiveness of two-dimensional and three-dimensional graph layouts are mixed. A general observation is that 3D layouts of large graphs, individual objects are often occluded and therefore barely recognizable, and orientation is sometimes intricate.

So called 2.5-dimensional visualizations show three dimensional objects arranged on a two-dimensional surface, similar to landscapes. This approach combines the good orientation and overview of 2D visualizations with a high information density, and exploits the same perceptual abilities that we use in our physical environment. Information retrieval systems based on the landscape metaphor mostly visualize data that has different characteristics than software structures, for example non-hierarchical data. In Software World [KNI00] an urban metaphor was used to visualize software systems. However, only few software entities are shown and the graphical representations are very simple. Neither the use of advanced visualization techniques nor the scalability to programs of realistic size is addressed in these works.

The DASADA (Dynamic Assembly for Systems Adaptability, Dependability, and Assurance) [DAS98]research program funded by the Department of Defense is divided into two interconnected infrastructures. A probe infrastructure is responsible for extracting meaningful events from the execution of a working component-based software system. A gauge infrastructure processes and delivers these events to specific gauges programmed to listen for events. A primary aim of DASADA is to develop a partnership between gauges and probes to evaluate a software system. However, gauges are carefully designed to observe specific situations and report results back to system administrators. The visualization of events can provides a standard means for observing the behavior. There are several important benefits of such a monitoring infrastructure. First, the probe infrastructure decouples the model (of the system) from the realization (implementation) of the system being monitored. Second, the approach is language independent. Third, it allows for exploration and customization, as different visualizations are activated in real or near-real time.

3.2.5 Traceability tools

A number of requirement traceability approaches and tools have been cited in both the literature and as by-products in the industry. These tools provide essential traceability links among various software artifacts to support software maintenance activities, like component substitution Some tools such as Teamwork/RQT [CAS93] provide capabilities to create different types of links. These traceability links include:parent and child relationships, functional hierarchies, definition of keywords and attributes to requirements and other system artifacts, ad-hoc and predefined querying, requirements extraction from documents, customized report generation, and maintenance of information about allocation of requirements to system components or functions.

Lindvall and Sandahl [LIN98] present a traceability approach based on domain knowledge to collect and analyse software change metrics related to impact analysis for resource estimates. However, their work does not consider automated *concept location*. They relate some change requests to the impacted code in terms of classes but no requirements and test cases are involved. Lange et al [LAN01] constructs a repository to handle maintenance tasks that link the code to testing and concept models. His concept model however is too general with respect to the requirements, business rules, reports, use cases, service functions and data objects. Bianchi *et al.* [BIA00] introduce and experiment with several examples of traceability link using ANALYST tool, with the aim of assessing how effectively these links support impact analysis in object-oriented environments and what effects they produce on the accuracy of the maintenance process.

3.2.6 Knowledge discovery tools

Program comprehension is an inherently difficult and essential problem because of the challenges in modeling domain knowledge and establishing the relationships between the domain model and source model. The discovery and utilization of domain knowledge in source code plays in a comprehension context an essential role. Tools that facilitate searching on source code, for instance `grep` and its variants, scruple [22] and `tksee` [30], allow the end-user to reverse abstract the information represented in the Software Landscapes, using both visual and textual data found in the source code and the diagrams. The Searchable Bookshelf is an extension of the Software Bookshelf as constructed by Finnigan et al [FIN97]. Searches are specified using GCL, a query language from information retrieval, designed for use with structured and semi-structured texts such as source code. GCL is distinguished by its support for queries that reference both structure and content, and by its uniform handling of structured data, so that diagrams, source code, and documentation can be searched using the same interface.

DM-TAO [BIG94] adopts a very complex *a prior* semantic network as the domain model, in which each domain concept is represented as a node and relationships between nodes are represented as explicit named links. The information associated with each concept includes: the typical features that characterize the concept, its relationships to other concepts in the domain, relevant informal knowledge – such as the terminology likely to be used by a programmer when referring to this concept in code, the syntactic and/or conceptual context this concept is likely to occur in, etc [BIG94]. Obviously, to construct such domain model needs tremendous effort, and it is unclear to us whether it is practical to provide such model for software maintainers.

In contrast, the domain model of the LaSSIE system [DEV91] is relatively simple, which is a formal ontology that captures major concepts of the problem domain, such as objects, actions, users, etc. Users of LaSSIE can query the domain model using declarative Description Logics expressions, or through a natural language interface. DM-TAO uses neural network technology to automatically recognize domain concepts in source code. LaSSIE system establishes such relationships manually, and thus is not feasible for large scale software systems or for time-pressured maintenance tasks

The LaSSIE system is good at faithfully and completely deriving concepts within small scale programs due to the fact that the relationships between domain and implementation have to be manually maintained. The DM-TAO system on the other hand can easily manage large scale programs but suffers from imprecision and therefore, is not trustworthy. However, constructing the domain model in DM-TAO is very difficult due to its complexity. Several approaches [LI_01, GOL02, and MAR04] have been proposed to reduce the complexity of Biggerstaff's domain model. Li et al [LI_01] proposed a simplified semantic network as the representation of domain knowledge, and each concept in the semantic network can find its counterpart in source code during a semi-automatic process. Gold et al [GOL02] also proposed a simplified domain model and applied similar algorithms as in Biggerstaff's approach.

Marcus et al [MAR04] proposed a novel approach that uses an Information Retrieval model – Latent Semantic Indexing (LSI) to automatically retrieve domain concepts in source code. The extensions of LaSSIE system [SEL94, WEL97] are mostly focus on enhancing the expressiveness of code model [WEL97] and improving the performance of LaSSIE system [SEL94].

The difficulty of discovering domain knowledge in source code is a manifestation of the general problem of software documentation – the same information (i.e. domain knowledge) is being stored in different software artifacts [WEL97] and the linkages between these artifacts are difficult to identify and maintain. As a result, software documents are frequently not up-to-date, and sometimes, contain incorrect information. Research in the area of traceability links [ANT00, MAR02] address this problem by trying to establish links between source code and software documents using Information Retrieval [KOW97], in an attempt to bridge the gap between domain knowledge and source code.

A study of LaSSIE [DEV91] showed that the integration between the domain and code models needed to be made manually making this approach time-consuming and difficult to maintain. In particular, since domain models change over time as new features are added to software systems. Maintainers start to lose faith in the domain model, as a result, the usage deteriorated. The code model, though very simple, is used therefore more frequently than the domain model

3.2.7 Data reverse engineering tools

Alborz is a user-assisted reverse engineering tool designed for use in analyzing and recovering software architecture in the form of cohesive modules and subsystems [SAR01]. The tool's operation is based on techniques taken from the areas of data mining, pattern matching, and clustering. The tool user defines a graph-based architectural pattern of system modules (subsystems) and their interactions based on domain knowledge, system documents, and tool provided clustering techniques. Through an iterative recovery process, the user constrains the architectural pattern, and the tool provides a decomposition of the system's entities into modules or subsystems that satisfy the constraints. The software system employs some form of a *database* to make its data persistent. Thus, there exists some form of database schema providing a static description of the data inside the database.

3.2.8 Tools supporting Software Evolution

The hardware on which the software runs is constantly subject to evolution, change and adaptation. To further complicate the situation, most programs do not reside directly on top of the hardware infrastructure but instead interact with an operating system, itself a software program. Programs themselves are also subject to the same evolution, change and adaptation that the hardware undergoes. The possible combinations and ways two programs can interact leads to a situation where the possible execution paths grows exponentially with the system size. One possible way to overcome this combinatorial exploration is by utilizing all electronic artifacts that are produced during the software development lifecycle. Information stored in versioning systems can be extracted and then correlated with other resources such as Bug tracking software and the communication infrastructure used by developers. From this information, statistics can be gathered and visually formatted to allow developers to find patterns and trends in the evolution of a software system. By using this information to create a higher level of abstraction when viewing the evolution of software, it is anticipated that new insights and practices can be developed. A survey reviewing 23 papers that deal with various perspectives and work on the extraction, correlation and visualization of information generated during software development can be found in [CHU05].

4 Processes Supporting Component Substitution

There exists a significant body of research on the development and use of process models to support the forward engineering of software systems. In the last couple of years, there was a trend towards integrating component based software development in these processes and adopt them to meet the requirements specific to component based software engineering. Common to most of these process models is their focus on the forward engineering (development) part of software systems and either none or only very limited consideration was given to maintenance aspects in these process models. The following section reviews some of the major process models that consider maintenance being part of a process model.

4.1 Maintenance Life Cycle and Process models

As large systems have proliferated and aged, the special needs of the operational environment have begun to emerge. Historically, the software lifecycle has usually focused on development. However, so much of a system's cost is incurred during its operational lifetime that maintenance issues have become more important and, arguably, this should be reflected in development practices and process models supporting the maintenance activities. Systems are required to last longer than originally planned. Inevitably, the percentage of costs going to maintenance has been steadily climbing. In this section several process models and software life cycle models for software maintenance activities will be surveyed. It should be noted that component substitution can be seen as a specific instance of a software maintenance activity. The general issue of modeling and supporting the processes involved in the maintenance of legacy systems is addressed in the literature from two different directions.

(1) The issue was addressed from a program comprehension view point, since program comprehension represent a major factor of software maintenance. Program comprehension comprises anywhere between 40-70% of the total effort and cost involved in the overall maintenance process. Over the last decade, several comprehension models where proposed in the literature, including bottom-up, top-down and integrated program comprehension models. Common to all of these models is the fact that they try to analyze and model the way programmers understand existing software systems. These comprehension models build often the general frameworks for tool developers to derive new techniques and tools to support a particular comprehension model. The reader is referred to the following articles for a more detailed coverage of these comprehension models [SEI06].

(2) Another approach is to make software maintenance an integrated part of a total system life cycle/process model. This was for example suggested by the Software Engineering Institute (SEI) in 1996, as part of their software technology roadmap for maintenance of operational systems (Figure 33).

Figure 33 *Total System Life Cycle [SEI06]*

The following is a brief summary of the major aspects of the total system life cycle models as suggested by the SEI [SEI06].

Complexity analysis. Before attempting to reach a destination, it is essential to know where you are. For a software system, a good first step is measuring the complexity of the component modules (see Cyclomatic Complexity and Halstead Complexity Measures). Maintainability Index Technique for Measuring Program Maintainability describes a method of assessing maintainability of code using those complexity measures. Test path coverage can also be determined from complexity measures, which can help in optimizing system testing (see Test generation and optimization).

Functionality analysis. Function Point Analysis describes the uses and limitations of function point analysis (also known as functional size measurement) in measuring software. By measuring a program's functionality, one can arrive at some estimate of its value in a system, which is of use when making decisions about rewriting the program or reengineering the system. Measures of functionality can also guide decisions about where to put testing effort (see Test generation and optimization).

Reverse engineering / design recovery. Over time, a system's code diverges from the documentation. This is a well-known tendency of operational systems. Another phenomenon that is frequently underestimated or ignored is that (regardless of the divergence effect), the information required to make a given change is often found only in the code. Several approaches are possible here. Various tools offer the ability to construct program flow diagrams (PFDs) from code. More sophisticated techniques, often classified as program understanding, are emerging. These technologies are implemented as tools that act as agents for the human analyst to assist in gathering information about a program's function at higher levels of abstraction than a program flow diagram.

Piecewise reengineering. If the system's known lifetime is sufficiently short, and if the evolutionary changes needed are sufficiently bounded, the system may benefit from a piecewise reengineering approach:

- Brittle, high-risk modules that are likely to need changes are identified and reengineered to make them more maintainable. Techniques such as wrappers, an emerging technology, are expected to aid here.

- For the sake of prudence, other risky modules are "locked," so that a prospective change to them can be made only after thoroughly assessing the risks involved.
- For database systems, it may be possible to retrofit a modern relational or object-oriented database to the system. Common Object Request Broker Architecture and Graphic Tools for Legacy Database Migration describe technologies of possible use here. Piecewise reengineering can generally be done at a lower cost than complete reengineering of the system. If it is the right choice, it delays the inevitable obsolescence. The downsides of piecewise reengineering include the following:
    - Platform obsolescence is not reversed. Risks arising from the platform's software are unchanged. If the original database or operating system has risks, the application using them will also.
    - Unforeseen requirements changes still carry high risk if they affect the old parts of the system.
    - Performance may suffer because of the interface structures added to splice reengineered functions to old ones.

Translation/restructuring/modularizing. Translation and/or restructuring of code are often of interest when migrating software to a new platform. Frequently, the new environment will not support the old language or dialect. Restructuring/modularizing, or rebuilding the code to reduce complexity, can be done simply to improve the code's maintainability, but code to be translated is often restructured first so that the result will be less complex and more easily understood. There are several commercial tools that do one or more of these operations, and energetic research to achieve more automated approaches is being done. Most often, it simply continues the existing problem in a different syntactical form. The mechanical forms output by translators decrease understandability, which is a key component of maintainability. None of these technologies is a cure-all, and none of them should be applied without first assessing the quality of the output and the amount of programmer resources required.

4.2 Component specific Software life cycle and Maintenance process models.

Component-based software engineering (CBSE) promised to revolutionize the way software is developed with the re-use of stable software components giving more functionality for less effort along with benefits in terms of time to market and reliability. Components-based development (CBD) should make it possible for developers to buy (COTS) in “expertise” or reuse (FOSS) from the marketplace in a form that is tested and reliable, pluggable and cost effective. Pluggable in this case means in a form that can be incorporated in the intended system with minor or no modification. The use of COTS software components has become increasingly prevalent in recent years. COTS components usage however, has resulted in a new set of problems that are not present when large systems are built and maintained using *custom-built* software. Because a COTS component is not specifically created for the application into which it is integrated, it may not meet all of the application's requirements. Furthermore, the source code for a COTS component is rarely made available to the buyer. Even if source code were available, determining how a COTS component will behave once it is integrated into a software system can still be difficult. In order to address these problems, one has created a methodology and a tool meant to aid developers when they attempt to integrate and substitute a COTS component with another component.

Many existing systems are implemented in several programming languages, including C, C++, and Fortran, with a large number of. software applications containing two or more programming languages. The *mixed-language* scenario addresses the need for models and techniques that can be used to analyze products that are implemented in a variety of languages and language types (procedural and object oriented). These systems may also include start-up files that configure the system at runtime based upon a set of script and data files. These systems may also have makefiles or build scripts that contain architecturally relevant information. How can all of the various components in several different languages and language types be modeled within a single reconstruction tool? What are the abstraction mechanisms for building architectural views from the source information from particular language types (procedural and object-oriented), and how can these be combined to produce architectural views that incorporate the different types of information? The challenge is to reconstruct the architecture of a system that is implemented in more than one language.

A number of reverse engineering activities focus on software architecture reconstruction. Kazman et al. [KAZ99] propose an iterative reconstruction process where the historical design decisions are discovered by empirically formulating/validating architectural hypotheses. They also point out the importance of modeling not only system information but also a description of the underlying semantics. Their approach is currently extended to include the reorganization of recovered assets into software product lines. Finnigan et al. [FIN97] propose a process that is supported by the Software Bookshelf: a toolkit to generate architecture diagrams from source text. Ding and

Medvidovic describe the Focus approach, which contrasts a *logical* (idealized, high-level) architecture with a *physical* (as implemented, as recovered) one. By applying refinement to the logical and abstraction to the physical architecture, the two are brought together incrementally. All the previous works differ in that they address a determined goal, concrete techniques, and a certain fixed sets of views to be reconstructed, rather than providing a more general reconstruction model. [DEU05]

Recovering modules and subsystems from existing software systems has proven useful in a number of ways and many methods to automatically or semiautomatically detect components have been published in the literature [ROU05, MIT02, and HAD02]. The abundance of published methods calls for frameworks to unify, classify, and compare them in order to make informed decisions. Girard and Briand introduce a process which synthesizes many methods to extract components from code. Koschke [KOS00] presents a comprehensive overview and a classification of existing component recovery methods. Girard and Koschke compare six published methods which recover abstract data types and objects. Koschke and Eisenbarth [EIS01]discuss the need for a standardized approach to compare component recovery methods and propose a framework to cost effectively run quantitative evaluation experiments [EIS01]

4.2.1 Component Comprehension Model

Andrews et al. [AND02] adapted the Integrated Comprehension Model for understanding software components. This model is shown in Figure 34. It has three levels: (1) Domain model, (2) Situation model, and (3) Program model. As with program comprehension models, developers can start building a mental model at any level that appears opportune, and switch between the three levels.

Figure 34. Integrated Model for Understanding Software Components [AND02]

Since developers usually do not have access to the source code for a COTS component, the understanding process is driven mainly by domain and design knowledge. Knowledge at the program model level is limited to interface specification of the component. Interface information may not be neatly packaged as an interface specification, but is extracted from various locations. Knowledge about the interfaces together with domain and situation models is used to understand COTS structure and behavior. At the domain level, the component’s external structure and behavior are determined. Application requirements are matched with component capabilities (see Figure 35).

A *source view* is a view of a system that can be extracted from artefacts of that system, such as source code, build files, configuration information, documentation, or traces. Some existing source views are at such a detailed level of granularity that they are not generally considered to be architectural views. For instance, the source view may cover abstract syntax trees and control flow graphs. A *target view* is a view of a software system that describes

the as-implemented architecture and contains the information needed to solve the problem/perform the tasks for which the reconstruction process was carried out.

A *hypothetical view* describes the architecture of the system, but perhaps not accurately. It can be a reference or a designed architecture used to check conformance of the implemented architecture to a norm. It can be a postulated architecture, describing the current understanding of the architecture of a system, and used to guide the reconstruction. This view is typically created by interviewing the system experts or by examining the existing documentation.

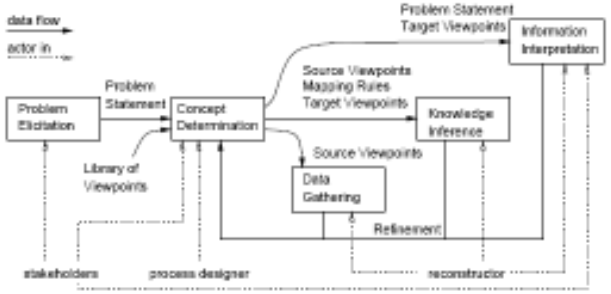


Figure 35. Interaction during reconstruction design [AND02].

4.2.2 COTS

One of the major challenges to conduct impact analysis (IA) for COTS based component systems is the need to have a process in place to support IA. It is necessary to document the process by which a system is developed, *as it is developed*, if it is going to be possible to analyze the potential impacts of future changes.

The *binary components* scenario covers architecture reconstruction using binary component descriptions [MIT01]. The software industry is quickly moving toward systems based on commercial components. A component in this context has three characteristics: it is produced by a vendor, who sells the component or licenses its use; it is released by a vendor in binary form; and it offers an interface for third-party integration [WAL02]. Existing architecture reconstruction methods abstract from the source code. Reconstructing software architecture in binary component settings is heavily dependent on the quality of the component interface descriptions. In addition, the detail in these descriptions may vary from vendor to vendor [REI04].

The situation where only binaries (and some moderately telling natural language documentation) are available seems to be the poorest among the options to be considered. However, this is quite common for people working with COTS. Binaries can be executed and it seems fair to assume that the rudimentary documentation that comes with them tells at least something about the environment needed and the functionality claimed for this component. The latter has to describe at least how input is to be presented to the component. If even this simple assumption is not warranted, we have to accept that the piece at hand is useless and we better don't waste time on it. If one is able to (systematically) execute a component though, this will lead at least two forms of complementary representation: the binaries, which are hard to understand in their materialized form, and test data in the form of tuples of input-output value combinations. One might use these data tuples as alternate representation from which it might be possible to infer the components semantic in a more compact form. Thus, the input-output tuples will constitute the basis on which the maintainer will form hypotheses about the component. These hypotheses about software need to be verified, irrespective on which grounds they have been established. In conventional forward engineering, one writes code and verifies the hypothesis that the code existing is the code we wanted by means of testing, in the case of comprehending the binaries it is the reverse.

A typical example for such challenge is an organization which is developing a Web-based application consisting of a Web server from one vendor and a database from another. The following example was introduced in [MIT01] and illustrates the major process parts needed to support the maintenance of COTS systems. In addition, two other commercial components must be integrated into the system. The organization's software architects want

to define the entire software architecture and understand what "glue parts" the organization must develop between the components (such as forms and data transformations). The component interfaces and partial architectural descriptions (e.g., the Web server and database architectures) are available. In the source code for the COTS components is not. There is a clear need for methods and tools to support this type of architecture reconstruction from binary components. The solution addresses a couple of open research issues in the architecture field. For example, how does one sufficiently describe components to be able to extract a software architecture? How does one know that assembled components satisfy the desired software architecture? Furthermore, how trustworthy are component descriptions? Assembling commercial components to achieve functional quality goals is a difficult task in the current component market. A commonly used approach is to build small toy examples to explore deficiencies so as to mitigate the risks for real products. Architecture reconstruction from commercial component descriptions could help architects detect the overall product structure and the dependencies among components. Effective maintenance and management of COTS-based systems depend on the ability to identify activities of the maintenance and management personnel [CLA98]. Once these activities have been identified, strategies can be developed to facilitate these activities. COTS-based maintenance and management, although similar in many respects to maintaining custom-built systems, have qualitative differences [MIT01].

Component reconfiguration. Reconfiguring components is the act of replacing, adding and deleting components within the system. Reconfiguration occurs for many reasons, perhaps the most common being the frequency with which commercial product vendors release updated versions of their software. It is not uncommon for each product to be upgraded two or three times per year. Often, system integrators are forced to replace older product versions with the upgrades in order to fix bugs or improve functionality. Other reasons for reconfiguring the components are to replace aging components with better products from competing vendors, or to add and delete products as the functional requirements of the system evolve. Reconfiguring the components is an expensive activity requiring the integrators to go through a complete release cycle including product evaluation, testing, design, integration, and system regression testing (Figure 36).

Troubleshooting and repair. All systems fail and COTS-based systems are no different in this respect. However, with COTS-based systems maintenance and management personnel generally cannot look inside components when trying to isolate the cause of the failure. Information must be gathered by experimenting at the edges of the components. Identifying the source of the fault requires running a series of experiments to determine the product or products causing the problem. Identifying and fixing the fault is no longer an activity performed solely by the system builders. Having used third-party products, system builders must now work closely with the support staff of the product suppliers, and with the general product user community. Where faults involve complex interactions involving sets of products from different vendors, many different organizations may be involved in the troubleshooting and repair of the system.

Configuration management. For COTS based systems, configuration management is done at the level of products rather than at the level of source code. Issues that maintainers must address include: change history for each individual product; availability and support level provided by the product vendor; management of configurations of the COTS-based system that are installed at each deployed site; compatibility requirements and constraints between sets of products; and licensing issues associated with each product.

Testing and evaluation. Testing and evaluating of COTS products is an ongoing activity during maintenance. Versions as well as new products must be evaluated for inclusion within the system and products must be tested during operational use.

Tailoring user level services. COTS products provide a generic functionality that can be used by many applications and organizations. System integrators must customize and tailor this functionality to satisfy the local operational requirements that are unique to the end-user organization. Successful systems are those that can be quickly modified and tailored to meet evolving user requirements. For COTS-based systems, tailoring involves an ongoing process of customizing and configuring products, adding new components to the system, and combining services of multiple products in novel ways. Since integrators do not have access to product source code, this must be done through gluing products together to provide enhanced functionality and using vendor supported tailoring techniques to customize the products.

System monitoring. System managers and maintainers must continuously monitor a system during its ongoing operation. This must be done to measure performance and resource usage, watch for failures, and determine user behavior. Because COTS software is black box, with limited visibility into internal behavior, monitoring for maintenance purposes can be difficult to do effectively.

Maintenance activity	Description
Component reconfiguration	Updating product versions, replacing COTS products with similar products, adding/deleting products
Troubleshooting	Identifying causes of failures among sets of COTS products, developing workarounds with the products, liasing with the COTS product maintainers
Configuration management	Tracking versions of different COTS products, tracking deployment configurations, determining compatible versions of products
Testing and evaluation	Testing new product versions as they become available, within the context of the system into which they will be integrated
Tailoring user level services	Enhancing the services available to the end user by configuring COTS products, combining services of multiple products, etc.
System monitoring	Monitoring different aspects of system behaviour, such as communication, resource usage, process invocation, etc.

Figure 36 Maintaining COTS-Based Systems [MIT01]

4.2.3 Related work[PIN03]

Software components are of primary concerns in the field of reverse engineering and there exist several tools and techniques that address the extraction of software components from available information (e.g. source code). However, these approaches don't take into account the information and architectural characteristics that are implicated by a particular application domain such as COM/COM+ component-based three-tiered applications. For example Koschke [KOS99] describes an approach that focuses on the detection of atomic components by integrating the user into the detection cycle. Atomic components are groupings of subprograms, types, and global variables, and can be viewed as cohesive logical modules. In contrast, one views components as deployable units that are self-contained. Sneed et al. describe a tool capable of analyzing very large applications. It uses a relational database for storing both the requirements specification and also the created implementation model. A mapping between records that represent particular requirements and those that represent implementation models is established in the database, too.

Subsequently, the database can be queried by SQL statements to find out about various system properties. Andrews et al. [AND02] identify COTS component comprehension as a specialized activity within software comprehension. They build a combined comprehension model for COTS components based on a domain model, a situation model, and a program model and evaluate how different component based software development approaches fit with this model. The focus in this research is on abstract comprehension models without providing a concrete implementation. Korel [KOR03] deals with COTS components, too. He treats components as black-box entities and uses interface probing as primary tool for obtaining component properties. To realize interface probing, a developer has to design a set of test cases that are executed on the component. The results can be evaluated and interface probing can be applied iteratively.

While black-box understanding methods can be applied when no source code is available, they are limited with respect to the number of properties they can extract. Java applets are specific Java components that can be started in web browsers. Korn et al. have developed Chava that supports the analysis of Java applets [KOR99]. They combine source code analysis with bytecode analysis techniques. Bytecode analysis can reveal some information about Java classes but cannot show the full picture since the Java compiler removes information.

4.3 Traceability and Recovery of Traceability Links

The issue of recovering traceability links between code and free text documentation, is not yet well understood and investigated, and very few contributions were published in the past 20 years. The traceability issue becomes an even more important factor for component substitution. Since components are not typically developed within an organization, it is essential for a maintainer not only to have access to these different artifacts related to a particular component/system but also to be able to establish links among these different artifacts and trace them. A number of related papers are in the area of impact analysis. For example, Turver and Munro [TUR94] assumed the existence of some form of ripple propagation graph describing relations between software artifacts, including both code and documentation, and focused on the prediction of the effects of a maintenance change request, not only on the source code, but also on the specification and design documents. Boldyreff et al. presented a method for the identification of traceability links of high-level domain concepts, using all available maintenance information and starting from a different view of traceability respect to IEEE definition.

Antoniol et al. [ANT00] presented a method to establish and maintain traceability links between code and free text documents. The method exploits probabilistic information retrieval techniques to estimate a language model for each document or document section, and applies Bayesian classification to score the sequence of mnemonics extracted from a selected area of code against the language models. The same method was applied to recover traceability links between the functional requirement and the Java source code, extending and validating the previous results on a more complex and difficult case study. The investigation was then extended to vector space models, to compare different models families and to assess the relative influence of affecting factors.

5 Recommendations and Conclusions

5.1 Component Substitution Process Models – A Critical View

One of the major challenges for component substitution is to provide a process that supports the comprehension and maintainability (substitution) of components. At the current stage, research focuses on the adoption of traditional comprehension models and software maintenance process models to support component substitution. On a first look, this approach seems to be promising, since some of the ground work has already been laid in existing models. However, one of the most common misconceptions is that one can address the problem by creating a general process model with the goal "one process model will fit all situations" approach. Component substitution is a multi-dimensional problem domain, where maintainers are faced also with challenges which are not only based on one dimension but can sprawl across several dimensions. These challenges include technical aspects, like the abstraction level, the type of component, as well as the type of substitution and the environment in which the substitution is being performed. Furthermore, the process has to consider also other aspects not directly related to the component, e.g. maintainers' expertise and understanding of the system, the availability of tools and techniques to support both the acquisition and interpretation of knowledge relevant to the substitution process, as well as support for performing and validating the substitution. Furthermore, one will have to deal with incomplete, inconsistent information resources, e.g. documentation versus source code. The process will significantly vary for example for COTS or FOSS components, due to the fact that the information that can be extracted and provided is completely different for these two categories.

Therefore, before one can define a component substitution process, it will be essential to provide a detail setting in which the substitution is going to be performed. The goal of these settings has to be to restrict the problem domain and therefore, the substitution space, to reduce its complexity and provide substitution categories that form the basis for deriving more problem (setting) specific substitution processes.

Another major challenge for a maintainer or programmer is to incorporate different knowledge resources in the comprehension/substitution process. This will improve the general understanding of the system, its components, their interactions and the impact of the change caused by a particular component substitution.

Any attempt to derive a full-automatic comprehension substitution process based solely on reverse engineering will fail, due to the fact that it is impossible to recover all the domain expertise and functional and non-functional requirements from existing source or bytecode that influenced the original design or implementation. Both program comprehension and component substitution will have to take into consideration the current understanding a maintainer has of the system under investigation. It will have to support the acquisition of new and additional knowledge, as well as facilitate some type of integration of knowledge, information needs, tools, techniques and

information provided. Program models or better component substitution models will have to reflect these issues by providing a representation that will closely match both a user's mental model of the system as well as the ability to reason about knowledge in these models. The process support has to go beyond simple querying and browsing. Rather, it will require some inferring of knowledge to provide more meaningful abstractions that support the substitution process.

Another open issue is that even with incorporating user expertise in the comprehension/substitution process, most of the existing models and processes are passive in their nature. Rather than actively helping the maintainer to acquire knowledge, they rely on the maintainer to initiate or follow a given model or process.

5.2 Concluding Remarks

Component substitution typically involves conflict identification, component adaptation, and implementation of connections between components. Usually, an existing component does not exactly match the requirements of a new system or the component to be replaced. Component adaptation is required in these cases in order to resolve the conflicts (or mismatches) between an existing component (the component to be replaced) and the requirements that the new component (substitute) has to match. There exist several major challenges with respect to component substitution. The focus of this state of the art report on component substitution is on the identification and comprehension of the system, the component to be substituted and the information and techniques that have to be applied to perform the substitution process. The substitution process can be seen as a multi dimensional problem domain, with each of these dimensions being summarized. These different dimensions are used to categorize, information needs, techniques, tools and processes involved in the component.

Software component substitution is not only a multi dimension and complicated process, due to the different levels of abstractions, the variety of systems and the user expertise one has to consider, but it is also not well defined. The component substitution process itself depends clearly on the environment and setting in which it is performed. Some of this questions which arise even before the substitution process is initiated:

- Is one dealing with COTS or FOSS?
- What type of substitution should take place (FOSS replacing COTS, COTS replacing COTS, FOSS replacing FOSS?)
- What is the experience of the domain expert, maintainer, management in charge of the component substitution?
- What resources are available? This includes issues like documentation, interface standards, etc.
- What is the overall goal of the substitution? Adding new functionalities, enhance existing functionalities, etc.
- What is the more specific goal of the substation process? Improve quality, security, reusability, provide independence from software suppliers.

A number of reverse engineering activities focus on software architecture reconstruction These approaches are an iterative reconstruction process, where the historical design decisions are discovered by empirically formulating/ validating architectural hypotheses. The software comprehension process by itself can be supported by a variety of techniques and tools. Most of these tools and techniques originate from traditional reverse engineering and program comprehension techniques. One of the challenges, especially for the substitution of components at a higher level, is to gain on the one hand sufficient insights to understand the structure and behavior of the existing system, and the component to be substituted. On the other hand, one has to be able to extract high level views, functional requirements and features and be able to trace them to the source code to be able to determine the risk, the impact of change, the adequate techniques and tools to support the substitution process, and to validate the successful substitution.

Due to the fact that most of these techniques are originating from the reverse engineering community, they are focusing mostly on the low to mid level system abstraction (statement/class level). For the comprehension of larger components, mainly at the package, subsystem or system level, higher level of abstraction is required to provide effective support during the component substitution process.

The following list of tables provides a brief summary of the tools and techniques covered in this report. A more detailed discussion on these techniques can be found in chapter 2 of this report.

Table 22 COTS – Statement execution level

Type of Component: COTS **Abstraction level : Statement, executed statement**				**Section: 2.1.1**	
Technique	Information need (input)	Information provided	Application	Static	Dynamic
Decompilation	Bytecode	Assembly code, Java code (decompiled)	Flow analysis Dependency analysis	Y	N
Instrumentation (virtual machine), bytecode	Bytecode	Execution traces	Dynamic flow and Dependency analysis	N	Y
Flow analysis (mainly control flow), Slicing	Decompiled assembly or Java Code	Control and data flow, limited slice	Flow analysis Dependency analysis	Y	Y

Table 23 FOSS – Statement source code level

Type of Component: FOSS **Abstraction level : Source code statement**				**Section 2.1.2**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Parsing, fact extraction, lexical analysis, queries, token analysis, strings and token matching	Source code, language parser	Facts, AST tokens, strings	Lexical analysis, queries, fact extraction flow analysis, token matching D query support (semantic, lexical) basis for most comprehension and reverse engineering tools	Y	N
Dependency analysis, semantic analysis, slicing, impact analysis	AST, tokens,	Control and data flow, limited slice	Dependency analysis, slicing, flow analysis impact analysis	Y	Y
Source code model	Source code, AST, syntactical/lexical analysis	Persistent storage (relational model, object-oriented model or ontology)	Basis for most reverse engineering tools	Y	N/Y
Tracing, profiling, monitoring	Bytecode, AST, source code	Execution traces	Dynamic flow and dependency analysis, profiling	Y	Y
Statement metrics	Source code, dependency analysis	Coverage, identifier complexity	Grouping, comprehensibility, profiling, maintainability	Y	N/Y

Table 24 COTS – Function/Class level

Type of Component: COTS **Abstraction level : Function/Class level**				**Section 2.2.1**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Decompilation	Bytecode	Assembly code, Java code (decompiled)	Flow analysis Dependency analysis	Y	N
Flow analysis Dependency analysis (mainly control flow), slicing	Bytecode Decompiler (Assembly or Java Code)	Control and data flow analysis, limited slice	Flow Analysis Dependency analysis	Y	Y
Instrumentation (Virtual machine), bytecode	Bytecode	Execution traces, call sequence, metrics	Dynamic Flow and Dependency analysis Call dependencies	N	Y

Table 25 FOSS – Function/Class level

Type of Component: FOSS **Abstraction level : Source code of Function/Class level**				**Section 2.2.2**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Design pattern recovery	Source code, program model language parser, pattern description	Identified design patterns	Comprehension, grouping, domain knowledge	Y	Y
Metrics	AST, program model, call dependencies	Different coupling, cohesion measurements	Maintainability, impact analysis	Y	Y
Slicing	Program model, source code, AST, control and dependency graph	Slice (executable, non-executable)	Comprehension, impact analysis, testing	Y	Y
Visualization	AST, program model	Abstract views, UML class model, sequence diagram.	Comprehension	Y	N/Y
Dependency Analysis	Source code, program model, control and data flow	Call-graph, call dependencies	Grouping, interaction	Y	Y

Table 26 COTS – Feature/Component/Package level

Type of Component: COTS **Abstraction level : Feature/Component/Package**				**Section 2.3.1.**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Protocol recovery	Static and Dynamic traces, profiler, domain knowledge,	Dependency analysis, Communication dependencies	Comprehension, impact analysis	Y	Y
Protocol validation	Static and dynamic traces, automata representation	Verification	Validating/verification that component protocols match before/after substitution	Y	Y
Wrappers/Glue Code	Component, interface	Standard interface, Access and Integration wrapper	Adaptation of legacy system, hooks, minimize change impact facilitates information and data exchange	Y	N
Component tailoring	Scripting, macro language	Plug-in, etc	Component enhancement	Y	N
GUI Ripping	GUI, executable COTS, test cases	GUI hierarchy	Reverse engineering of COTS, adapting legacy applications to new systems, Feature extraction	N	Y
Component metrics	Component, interface, function parameters	Meta information, component customizability, parameter completeness	Prediction of maintainability and substitution	Y	N
Compositional Component Adaptation	Standardized interfaces, dynamic information	Abstraction, separation of concerns	Maintainability, comprehension	Y	Y
Technique	Information need (input)	Information provided	Application	Static	Dynamic

Table 27 FOSS– Feature/Component/Package level

Type of Component: FOSS **Abstraction level : Source code for Feature/Component/Package**				**Section 2.3.2.**	
Technique	**Information need**	**Information provided**	**Application**	**Static**	**Dynamic**
Change impact analysis, ripple effect, traceability, dependency analysis	Static and dynamic traces, program model, domain knowledge, system model	Ripple effect analysis, trade-off analysis, validating substitution	Comprehension, impact analysis, maintenance, protocol verification	Y	Y
Concept analysis, grouping techniques, feature extraction	Program model, domain knowledge, source code, program traces	Features, logical grouping and clustering	Comprehension, feature extraction, substitution, testing, refactoring, feature analysis	Y	Y
Behavioral analysis	Tracing, domain knowledge	Dependency analysis, communication dependency	Comprehension, component interaction analysis	N	Y
Aspect oriented	Source code, binary (Java), program model	Separation of concerns	Refactoring, tracing	Y	Y
Data reverse Engineering	Schema, database, domain knowledge	Grouping , feature extraction, associations, dependency analysis	Program comprehension, maintenance	Y	Y
Software Visualization	Source code, traces, program model, domain knowledge	Abstraction, views	Structural comprehension, Interaction	Y	Y

Table 28 COTS – Subsystem/Architecture level

Type of Component: COTS **Abstraction level : Subsystem/Architectural level**				**Section 2.4.1.**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Testing based	Test suites, traces	Functional and non-function properties, e.g. security	Comprehension, impact analysis	N	Y
Comprehension of standard COTS	EFSM system model Input testing	State change, dependency analysis with libraries, etc	Verification, testing, comprehension	Y	Y

Table 29 COTS – Feature/Component/Package level

Type of Component: FOSS **Abstraction level : Subsystem/Architectural level**				**Section 2.4.2.**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Architecture extraction	Source code, domain knowledge, file names	Interaction among components	Comprehension, dependency analysis	Y	N
Architecture Reconstruction	Program model, source code, domain knowledge, scoping, visualization	High-level views	Validating of performed substitution	Y	N
Architectural Transformation	Program model task, domain knowledge, trace information	Architectural analysis, transformed architecture	Adaptation of architecture, legacy systems conversion	Y	N/Y
Architectural metrics	Program model, CVS data, package information,	Object/package dependencies	Evaluation, comprehension	Y	N
Visualization techniques	Program model, file, package information	Abstract view, package, 4+1 view	comprehension, impact analysis	Y	N
Impact analysis	Documentation, domain knowledge, program model, query language	Affected subsystem parts	Dependency analysis	Y	N
Dynamic analysis, logging, visualization	Log files, sampling, traces	Communication dependencies, interaction diagrams, behavioral models	Re-modularization, interactions analysis, comprehension	**N**	Y

Table 30 Document level analysis

Type of Component: Documentation **Abstraction level : Documentation**				**Section 2.5.**	
Technique	**Information need (input)**	**Information provided**	**Application**	**Static**	**Dynamic**
Domain knowledge retrieval	Domain expert, documentation, source code queries, domain model	Functional/non-functional requirements	Comprehension, architectural recovery, architecture analysis	Y	N
Data mining, knowledge discovery	Data, traces, data bases, logs, domain expert	Patterns, prediction, clustering	Discovery of knowledge that is not directly identifiable in the source code	Y	Y
Traceability	Domain model, program model, documentation, domain expert	Traceability from requirements (functional to source code	Maintenance, comprehension, verification	Y	N

As illustrated in the previous tables, there exists a variety of techniques, based on different information requirements, and providing different levels of analysis and support for the component substitution process.

Some of the major challenges in supporting the component substitution process are the following:

1. Defining a novel process that focuses specifically on the issues related to component substitution, rather than adopting existing process and comprehension models to support the substitution process. This is partially due to the fact the component substitution has some very specific requirements and goals that are not necessarily the main focal point in traditional architectural reconstruction and reverse engineering approaches.
2. There is a clear need of guiding users, maintainers during the comprehension process. This integration has to go beyond just integrating tools, services and their information. A typical example for this challenge can be found in existing CASE tools, like Rational. Rational supports all facets of the RUP process - tools are integrated and share information. However, the process itself is not clearly visible and supported by the tool. This example clearly illustrates not only the need for a process model, but also its integration in a model and tool support.
3. Techniques and tools supported by the process, accepted by users and supported by the infrastructure. This issue is often overlooked. In the past, there was a clear focus on detailed, semantic and syntactical analysis in the reverse and program comprehension community. One of the major drawbacks of these approaches were, that these tools were hard to use, not necessarily efficient, not integrated in the programming environment, not supported by the system administrator and often, due to the complexity of the underlying analysis, these techniques are limited to subset of environments and often inflexible in their use. This observation is also documented by widely use of grep as one of the most popular comprehension tools. Integrated in the OS, easy to use, quick response. There is a clear need for new, integrated (in the programming environment) query techniques that support a user mental model, support the validation of user hypotheses and are flexible enough to adopt to different comprehension tasks.
4. Documentation and domain experts have to play an important role in any component substitution and comprehension process. It would be illusive to expect a fully automatic comprehension substitution process. Furthermore, any substitution process will have to take advantage of knowledge that exists in the form of documentation, a maintainer's domain expertise, etc. It is the need to incorporate different knowledge sources and the ability to search, query and reason about these different knowledge resources that will provide the real benefit towards the component substitution process.
5. The scalability, navigability and abstraction provided by visual representations will remain a constant challenge in providing meaningful abstraction of information. However, visual representations have to go beyond the traditional view generation, and provide and support more flexible metaphors that not only incorporate some of the traditional analysis information, but also consider issues like layout, grouping, visualizing domain expertise and trying to capture and represent the current mental model and understanding of the system under investigation.

6 References

6.1 References in alphabetic order

[AHO79]	A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "awk - A Pattern Scanning and Processing Language". Software Practice and Experience, Vol. 9, No. 9, pages 267-280, 1979	73		Book
[AMI04]	Srivastava Amitabh, Eustace Alan, "ATOM a system for building customized program analysis tools:", ACM SIGPLAN notices (ACM SIGPLAN not.) ISSN 1523-2867, 20 years of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1979-1999. A selection 2004, vol. 39, no 4 (14 ref.), pp. 530-539	1		
[AMY01]	D. Amyot, A. Eberlein, "An Evaluation of Scenario Notations for Telecommunication Systems Development," 9th Int. Conference on Telecommunication Systems (9ICTS), Dallas, USA (March 2001).	40		IEEE
[ANA99]	Anaconda, http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/Anaconda/Welcome_en.html			tool web page
[AND02]	A. Andrews, S. Ghosh, and E. M. Choi. A model for understanding software components. In Proc. of the International Conference on Software Maintenance, pages 359–368, Montreal, Canada, October 2002. IEEE Computer Society Press\	9		IEEE
[AND98]	J. Andrews, "Testing using Log File Analysis: Tools, Methods, and Issues," ase, p. 157, 13th IEEE International Conference on Automated Software Engineering (ASE'98), 1998	25		
[ANT00]	G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation". In Proceedings of IEEE International Conference on Software Maintenance, San Jose, CA, 2000	21		
[ARN93]	R. S. Arnold, S. A. Bohner, "Impact Analysis - Towards a Framework for Comparison", Proc. Conf. Software Maintenan65ce, 1993, pp. 292-301.	65		IEEE
[AUE04]	Auer, Sören; Formal Concept Calculator, http://www.advis.de/soeren/fca/index.php			tool web page
[BAA03]	F. Baader et al., "The Description Logic Handbook", Cambridge University Press, 2003	538		BOOK
[BAI02]	Xiaoying Bai, Wei-Tek Tsai, Ke Feng, Lian Yu, Ray J. Paul, "Scenario-Based Modeling and Its Applications," WORDS 2002: 253-260.	13		
[BAL99]	Ball, Thomas, "The Concept of Dynamic Analysis", ACM Conference on Foundations of Software Engineering, Toulouse,France, September 1999, pp.216-234.	66		
[BAS01]	Basili, V.R. and Boehm, B., "COTS-based systems top 10 list", Computer , 34(5), May 2001, pp. 91-95.	77		
[BCL06]	http://jakarta.apache.org/bcel/index.html			Web page

[BCM03]	F. Baader et al., "The Description Logic Handbook", Cambridge University Press, 2003			Book
[BER94]	A. Bergstra and P. Klint. The ToolBus -- a Component Interconnection Architecture--. P9408, Programming Research Group, University of Amsterdam, Amsterdam, 1994.	25		
[BER97]	Bergeron, J., Debbabi, M., Erhioui, M.M., and Ktari, B. 1999. Static analysis of binary code to isolate malicious behaviors, In Proc. IEEE International Workshop on Enterprise Security.	17		
[BIA00]	Alessandro Bianchi, Giuseppe Visaggio, Anna Rita Fasolino: An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models. IWPC 2000: 149-			
[BIG94]	T. J. Biggerstaff, G. B. Mitbander, and D. Webster. "Program Understanding and the Concept Assignment Problem". Communications of the ACM, 37(5):72-83, May 1994.	167		
[BIN04]	D. Binkley, M. Harman. A Survey of Empirical Results on Program Slicing. Advances in Computers, Volume 62, 2004. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.			
[BIR67]	Birkhoff, G.; Lattice Theory, American Mathematical Society, Providence, 2nd Edition, 1967.			Book
[BLA01]	Sue Black, "Computing Ripple Effect for Software Maintenance," Journal of Software Maintenance 13(4), pp. 263-279, 2001.	6		IEEE
[BOH96]	S. Bohner and R. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996, pp. 1-26.	127		Book IEEE Press
[BOR01]	Francis Bordeleau, Jean-Pierre Corriveau, "On the Importance of Inter-Scenario Relationships," www.usecasemaps.org/urn/cascon01/scenarioRelationships.pdf	3		
[BRA99]	Brandenburg, F.J., Graphlet, Universität Passau, http://www.infosun.fmi.uni-passau.de/Graphlet			tool web page
[BRE00]	K. Breitman, J. C. Sampaio do Prado, "Scenario Evolution: A Closer View on Relationships," Proceedings. 4th International Conference on Requirements Engineering, 2000, pp. 95 - 105.	9		IEEE
[BRE00]	K. Breitman, J. C. Sampaio do Prado, "Scenario Evolution: A Closer View on Relationships," Proceedings. 4th International Conference on Requirements Engineering, 2000, pp. 95 - 105.	9		
[BRO00]	Lisa Brownsword, Tricia Oberndorf, Carol A. Sledge, "Developing New Processes for COTS-Based Systems," IEEE Software, vol. 17, no. 4, pp. 48-55, Jul/Aug, 2000.	70		
[BRU90]	K. Bruce and P. Wegner, "An Algebraic Model of Subtype and Inheritance". Advances in Database Programming Languages, Chapter 5, F. Bancilhon and P. Buneman, editors, ACM Press, 1990			Book
[BRU93]	B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analysis. In Conference on Object-Oriented Programming Systems, Languages, and	4		

	Applications (OOSLA93), Washington, USA, September 1993.			
[BUL02]	R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey, "Semantic Grep: Regular Expression + Relational Abstraction", In Proc of the 9th Working Conference on Reverse Engineering (WCRE'02), 2002	10		
[CAN00]	Gerardo Canfora, Jorg Czeranski, Rainer Koschke, "Revisiting the Delta IC Approach to Component Recovery," wcre, p. 140, Seventh Working Conference on Reverse Engineering (WCRE'00), 2000.	3		
[CAN98]	G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, Information and Software Technology Special Issue on Program Slicing, volume 40, pages 595--607. Elsevier Science B. V., 1998.	42		
[CAN99]	Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A. (1999a), 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', Workshop on Program Comprehension, pp. 136-143, Pittsburgh, IEEE Computer Society Press.	13		IEEE
[CAR99]	S. J. Cartiere, S. Woods, and R. Kazman, "Software architectural transformation," in Proc. 6th Working Conf. Reverse Engineering, Oct. 1999.	18		
[CAS93]	Casotto, A., 'Run-Time Requirement Tracing", IEEE INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN Bibliographic details 1993, , pages 350			
[CHE90]]	Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, "The C Information Abstraction System". IEEE Transactions on Software Engineering, 16(3):325-334, 1990	173		
[CHE93]	Cheng J., "Slicing concurrent programs a graph-approach", In Proceedings of the First International Workshop on Automated and Algorithmic Debugging (1993), P. Fritzson, Ed., Vol. 749 of Lecture Notes in Computer Science, Springer-Verlag, pp. 232-245, 1993.			
[CHU05]	D. Church. A survey of techniques for the recovery and observation of software evolution. Unpublished, 2004.	1		
[CIF97]	Cristina Cifuentes and Antoine Fraboulet, "Intraprocedural Static Slicing of Binary Executables", ICSM 199, pp.180	10		
[CLA98]	J Clapp, A Taub - MP 98B0000069, " A Management Guide to Software Maintenance in COTS-Based Systems", The MITRE Corporation, Bedford, MA, November, 1998 - mitre.org	12		
[CLO06]	Clover, "Frequently Asked Questions", http://www.cenqua.com/clover/doc/faq.html, retrieved September 2005.			
[CLO06]	http://www.cenqua.com/clover/			Web page
[CON03]	Concept Explorer Project, http://sourceforge.net/projects/conexp			tool web page
[DAI06]	http://pag.csail.mit.edu/daikon/			Web page
[DAS98]	DASADA , A Model for Designing Adaptable Software Components (22nd Annual international Computer Science and Application Conference (COMPSAC-98).			

	Pages 121-127, August 1998. Vienna, Austria. View Abstract)			
[DBS91]	P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, “LaSSIE: a Knowledge-based Software Information System”. Communications of the ACM, 34(5):36–49, 1991	266		
[DEK03]	Dekel, U. and Gil, Y. 2003. Visualizing class interfaces with formal concept analysis. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM Press, New York, NY, 288-289. DOI= http://doi.acm.org/10.1145/949344.949416	1		
[DEM99]	Serge Demeyer, Stphr, Ducasse and SanderTichPLLI("A Pattern Language for Reverse Engineering," Proceedings of th4th European Conference on Pattern Languages of Programming and Computing, 1999, Paul Dyson (Ed.), UVK Universittsverlag Konstanz GmbH, Konstanz, Germany, July 1999.	15		
[DEU04]	A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04).	4		
[DEU05]	A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04)	15		
[DEU99]	van Deursen, A.; Kuipers, T.; Identifying Objects Using Cluster and Concept Analysis, In Proceedings of the 1999 International Conference on, 16-22 May 1999, p. 246 -255.	85		
[DEV91]	P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: a Knowledge-based Software Information System". Communications of the ACM, 34(5):36-49, 1991	266		
[DON05]	Kim, Soo Dong, and, Park, Ji Hwan,"C-QM: A Practical Quality Model for Evaluating COTS Components ," Proceedings of International Association of Science and Technology for Development (IASTED) International Conference on Software Engineering (SE'2003), Innsbruck, Austria, PP.991-996, Feb. 10-13, 2003.	3		
[DOS99]	DOS Programs of the Darmstadt Research Group on Formal Concept Analysis, http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/DOS-Programme/Welcome_en.html			tool web page
[DUE92]	Duesterwald E., Gupta R., and Soffa M., “Distributed slicing and partial re-execution for distributed programs”, In Proceedings of the fifth workshop on Languages and Compilers for Parallel Computing, New Haven, Connecticut, 1992,.pp. 329-337,			
[DWY98]	Matthe w B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Proceedings of the 21st Intl. Conf.	221		

	on Software Engineering, 1999.			
[EBE02]	Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. Gupro - Generic Understanding of Programs. Electronic Notes in Theoretical Computer Science, (2):59–68, 2002.	3		
[EIS01]	Thomas Eisenbarth, Rainer Koschke, Daniel Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces," iwpc, p. 0300, Ninth International Workshop on Program Comprehension (IWPC'01), 2001	26		IEEE
[FEI98]	L. Feijs, R. Krikhaar, and R. C. Ommering, “A Relational Approach to Support Software Architecture Analysis”. Software Practice and Experience, 28(4):371-400, 1998	55		
[FEN99]	Fenton NE, "Software Measurement Programs", Software Testing & Quality Engineering 1(3), 40-46, 1999.			
[FER01]	Ferenc, R., Magyar, F., Beszédes, Á., Kiss, Á. and Tarkiainen, M. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001), pages 16-27. Szeged, Hungary, June 15-16, 2001. Published by University of Szeged.	16		
[FIN97]	Finnigan, P. J. Holt, R. C. Kalas, I. Kerr, S. Kontogiannis, K. Mueller, H. A. Mylopoulos, J. Perelgut, S. G. Stanley, M. Wong, K., The software bookshelf,IBM SYSTEMS JOURNAL, 1997, VOL 36; NUMBER 4, pages 564-593	163		
[GAM94]	E. Gamma, R. Helm, R. Johnson, J. Vlissides “Design Patterns – Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1994.			BOOK
[GAN03]	Gerald C. Gannod, Shilpa Murthy: Verification of Recovered Software Architectures. IWPC 2003: 258-265	1		
[GAR01]	Garg V. K. and Mittal N., On Slicing a Distributed Computation, In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS), pages 322-329, Phoenix, Arizona, USA, April 2001.			
[GAR01]	Marcela Genero, Jos Olivas, Mario Piattini, and Francisco Romero,2001,"Using metrics to predict OO information systems maintainability",13th International Conference Advanced Information Systems Engineering.	123		
[GOL02]	N. Gold, K. Bennett, "Hypothesis-based Concept Assignment in Software Maintenance", IEEE Proceedings of Software Engineering, Vol 149, Issue 4, 2002	15		
[GOR05]	Ian Gorton, Liming Zhu: Tool support for just-in-time architecture reconstruction and evaluation: an experience report. ICSE 2005: 514-523	1		
[GRA92]	J. E. Grass, "Object-Oriented Design Archaeology with CIA++". Computing Systems, Journal of the USENIX Association, 5(1):5-67, Winter, 1992	24		IEEE
[HAD02]	Haddox, J. M. Michael, C. C. Kapfhammer, G. M., "AN APPROACH FOR UNDERSTANDING AND TESTING THIRD PARTY SOFTWARE COMPONENTS", IEEE PROCEEDINGS OF THE ANNUAL RELIABILITY, AND	8		

	MAINTAINABILITY SYMPOSIUM, 2002, ISSU 2002, pages 293-299			
[HAM01]	Hamou-Lhadj, Abdelwahab, and Timothy C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, Markham, Canada, October 2004, pp.42-55.	4		
[HAR97]	Mark Harman, Sebastian Danicic"Amorphous Program Slicing",Proceedings of the 5th International Workshop on Program Comprehension (WPC '97), 1997,pp70	37		
[HIS99]	Hissam, S. A. and Carney, D. 1999. Isolating faults in complex COTS-based systems. Journal of Software Maintenance 11, 3 (May. 1999), 183-199.			
[HOG00]	Kathi Hoggshead Davis and Peter Aiken "Data Reverse Engineering: An Historical Survey" Proceedings of the 7th Working Conference on Reverse Engineering Brisbane, Queensland, Australia • November 23-25, 2000 • pages 70-78	12		
[HOL96]	R. C. Holt, “Binary relational algebra applied to software architecture”. Technical report, CSRI 345, University of Toronto, march 1996	26		
[IBR05]	Suhaimi Ibrahim, Norbik Bashah Idris, Malcolm Munro, Aziz Deraman: Implementing a Document-based Requirements Traceability: A Case Study. IASTED Conf. on Software Engineering 2005: 124-131	0		
[IEE00]	IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html			
[IEE90]	610.12-1990 (R2002) IEEE Standard Glossary of Software Engineering Terminology			Online only
[IEE98]	1219-1998 IEEE Standard for Software Maintenance			Online only
[JAI99]	Jain, A. K., Murty, M. N., and Flynn, P. J. 1999. Data clustering: a review. ACM Comput. Surv. 31, 3 (Sep. 1999), 264-323. DOI= http://doi.acm.org/10.1145/331499.331504	1153		
[JAL04]	JaLaBA - Online Java Lattice Building Application for Concept Lattices, http://maarten.janssenweb.net/jalaba/JaLaBA.pl			tool web page
[JAL04]	JaLaBA - Online Java Lattice Building Application for Concept Lattices, http://maarten.janssenweb.net/jalaba/JaLaBA.pl			tool web page
[JAV06]	http://www.csg.is.titech.ac.jp/~chiba/javassist/			Web page
[JOE06]	http://www.cs.duke.edu/ari/joie/			Web page
[KAZ96]	Rick Kazman, Gregory D. Abowd, Leonard J. Bass, Paul C. Clements, "Scenario-Based Analysis of Software Architecture," IEEE Software 13(6), pp. 47-55 (1996)	235		http://www.sei.cmu.edu/architecture/scenario_paper/
[KAZ96]	Kazman, Rick, 1995, Scenario-based analysis of	235		

	software architecture: Waterloo, Ont.,Canada, University of Waterloo, Computer Science Dept..			
[KAZ98]	R. Kazman and S.J. Carrière. “View Extraction and View Fusion in Architectural Understanding.” Proceedings of the 5th International Conference of Software Reuse, Victoria, B.C., Canada, June, 1998.	70		
[KAZ99]	Rick Kazman, S. Jeromy Carrière, Playing Detective: Reconstructing Software Architecture from Available Evidence, Automated Software Engineering, Volume 6, Issue 2, Apr 1999, Pages 107 - 138	138		
[KIS05]	Ákos Kiss, Judit Jász, Tibor Gyimóthy, Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables, Software Quality Journal, Volume 13, Issue 3, Sep 2005, Pages 227 - 245,	6		
[KLI03]	P. Klint, “How Understanding and Restructuring Differ from Compiling – A Rewriting Perspective”. 11th IEEE International Workshop on Program Comprehension, 2003.	10		
[KNE03]	Antje von Knethen, Mathias Grund, "QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces," Int'l Conference on Software Maintenance 2003, pp. 246-255.	3		IEEE
[KNI00]	Claire Knight, Malcolm Munro, "Virtual but Visible Software," iv, p. 198, Fourth International Conference on Information Visualisation (IV'00), 2000.			
[KNO05]	Jens Knodel, Isabel John, Dharmalingam Ganesan, Martin Pinzger, Fernando Usero,Jose L. Arciniegas, and Claudio Riva, ,"Asset Recovery and Incorporation into Product Lines" Proceedings of the 12th Working Conference on Reverse Engineering, 2005, Pages: 120-129			
[KOR03]	Bogdan Korel, Inderdeep Singh, Luay Tahat, Boris Vaysburg, "Slicing of State-Based Models," Int'l Conference on Software Maintenance, 2003, pp.34-43.	3		
[KOR87]	Korel, B. and Laski, J., “Dynamic program slicing”, In. Process. Letters, 29(3), pp. 155-163, Oct. 1988.			
[KOR90]	B. Korel, J. Laski, “Dynamic slicing of computer programs,” Journal of Systems and Software, 13(3), pp.187-195, 1990.	233		
[KOR92]	Korel, B. and Ferguson, R., “Dynamic slicing of distributed programs”, Applied Mathematics & Computer Science Journal, 2(2), pp. 199-215, 1992.			
[KOR99]	Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In Proc. Working Conference on Reverse Engineering, 1999	36		
[KOS02]	Rainer Koschke und Yan Zhang, Component Recovery, Protocol Recovery and Validation. In: 3. Workshop Software-Reengineering, Bad Honnef (10./11.Mai 2001), Fachberichte Informatik, Universität Koblenz-Landau, Nr. 1/2002, pages 73-76, Januar 2002.	4		
[KOS06]	Koschke, Rainer," What architects should know about reverse engineering and reengineering", In: IEEE/IFIP Working Conference on Software Architecture, IEEE Computer Society Press, November 2006, S. 4-10	0		
[KOS97]	Jean-Francois Girard, Rainer Koschke. A Comparison of	18		

	Abstract Data Type and Objects Recovery Techniques. Journal Science of Computer Programming, Volume 36, Issue 2-3, pp. 149-181, Elsevier, March 2000			
[KOS99]	Koschke, R. 1999. An Incremental Semi-Automatic Method for Component Recovery. In Proceedings of the Sixth Working Conference on Reverse Engineering (October 06 - 08, 1999). WCRE. IEEE Computer Society, Washington, DC, 256.	14		
[KOT05]	Gerald Kotonya, John Hutchinson: Analysing the Impact of Change in COTS-Based Systems. ICCBSS 2005: 212-222			
[KOW97]	G. Kowalski, "Information Retrieval Systems: Theory and Implementation". Kluwer Academic Publishers, 1997	198		Book
[KRA96]	Kramer, C. and Prechelt, L. 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96) (November 08 - 10, 1996). WCRE. IEEE Computer Society, Washington, DC, 208.	128		
[KRI01]	Krinke J., Static slicing of threaded programs. In Proceedings of the ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), pages 35-42.			
[KRI99]	R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, C. Verhoef, "A Two-Phase Process for Software Architecture Improvement," icsm, p. 371, 15th IEEE International Conference on Software Maintenance (ICSM'99), 1999.	24		
[KRI99]	Krikhaar, R. Software Architecture Reconstruction. Ph.D. Thesis, University of Amsterdam, June 1999.	49		
[KUI00]	Kuipers, T.; Moonen, L.; Types and Concept Analysis for Legacy Systems, Program Comprehension, 2000, In Proceedings of the 8th International IWPC, Oct-Nov 2000.	27		
[KUL00]	B. Kullbach, “Command Line GReQL: CLG”, University of Koblenz-Landau, German, 2000	1		
[KUL99]	B. Kullbach, and A. Winter, “Querying as an Enabling Technology in Software Reengineering”. In Proceedings of the 3rd EuroMicro Conference on Software Maintenance and Reengineering, 1999	27		
[KUN94]	David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, Cris Chen, "Change Impact Identification in Object Oriented Software Maintenance," Int'l Conference on Software Maintenance,1994, pp. 202-211.	87		
[KVA05]	Kvale, A. A., Li, J., and Conradi, R. 2005. A case study on building COTS-based system using aspect-oriented programming. In Proceedings of the 2005 ACM Symposium on Applied Computing (Santa Fe, New Mexico, March 13 - 17, 2005). L. M. Liebrock, Ed. SAC '05. ACM Press, New York, NY, 1491-1498.	8		
[LAI83]	P. N. Johnson-Laird, “Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness”. Harvard University, Cambridge, Mass., 1983.	14		

[LAM98]	W. Lam, V. Shankararaman, S. Jones, J. Hewitt, C. Britton, "Change Analysis and Management: a Process Model and Its Application Within a Commercial Setting," Symposium on Application-Specific Systems and Software Technology (ASSET-98), March 1998, pp. 34 - 39.	0		IEEE
[LAN01]	C. Lange, H. M. Sneed, and A. Winter, “Applying the graph-oriented GUPRO Approach in comparison to a Relational Database based Approach”. In Proc of the 9th International Workshop on Program Comprehension, 2001	8		
[LAS02]	assing, N., Bengtsson, P., van Vliet, H., and Bosch, J. 2002. Experiences with ALMA: architecture-level modifiability analysis. J. Syst. Softw. 61, 1 (Mar. 2002), 47-57. DOI= http://dx.doi.org/10.1016/S0164-1212(01)00113-3	22		
[LAW03]	James Law, Gregg Rothermel, Whole Program Path-Based Dynamic Impact Analysis,” Internationall Conference on Software Engineering, 2003, pp. 308-318.	19		IEEE
[LEE00]	M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," Computer Performance Evaluation: Modelling Techniques and Tools, 11th International Conference, TOOLS, 2000.	18		
[LEE96]	H. Lee, B. Zorn. BIT : Bytecode Instrumenting Tool. University of Washington (1996). http://www.cs.colorado.edu/~hanlee/BIT/index.html			Web page
[Li_01]	Y. Li, H. Yang, W. Chu, "Generating linkage between source code and evolvable domain knowledge for the ease of software evolution". In Proceeding s of the International Symposium on Principles of Software Evolution, Nov. 2001	2		IEEE
[LIN02]	Lindig, Christian; Concepts, http://www.st.cs.uni-sb.de/~lindig/src/concepts.html			tool web page
[LIN95]	Lindig, C.; Concept-based Component Retrieval, In Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors, Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, Montreál, Aug 1995, p. 21-25.	46		
[LIN96]	T. Lindholm and F. Yellin. The Java Virtual Machine. Addison-Wesley, 1996.			
[LIN97]	Lindig, C.; Snelting, G.; Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, Software Engineering, 1997., In Proceedings of the 19th International Conference, May 1997, p.349-359.	119		
[LIN98]	Mikael Lindvall, Kristian Sandahl," Traceability Aspects of Impact Analysis in Object-Oriented Systems", Published in Journal of Software Maintenance, No 10, pp 37-57, 998.			
[LUC00]	G. A. D. Lucca, A. R. Fasolino, and U. D. Carlini. Recovering use case models from object-oriented code: A thread based approach. In Proceedings of the 7th Working Conference on Reverse Engineering (WCRE’00), pages 108–117, 2000.			

[MAR04]	A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code". in the Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, pp. 214-223, November 9-12, 2004	8		IEEE
[MCK04]	P.K. McKinley et al., "A Taxonomy of Compositional Adaptation," tech. report MSU-CSE-04-17, Dept. Computer Science and Engineering, Michigan State Univ., 2004	5		
[MEM03]	Atif Memon , Ishan Banerjee , Adithya Nagarajan, GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, Proceedings of the 10th Working Conference on Reverse Engineering, p.260, November 13-17, 2003	15		
[MET00]	Mehta, A. and Heineman, G. T. 2001. Evolving legacy systems features using regression test cases and components. In Proceedings of the 4th international Workshop on Principles of Software Evolution (Vienna, Austria, September 10 - 11, 2001). IWPSE '01. ACM Press, New York, NY, 190-193. DOI= http://doi.acm.org/10.1145/602461.602507	1		
[MET02]	Mehta, A. and Heineman, G. T. 2002. Evolving legacy system features into fine-grained components. In Proceedings of the 24th international Conference on Software Engineering (Orlando, Florida, May 19 - 25, 2002). ICSE '02. ACM Press, New York, NY, 417-427. DOI= http://doi.acm.org/10.1145/581339.581391	15		
[MIT01]	Mittermeir, R. T., Bollin, A., Pozewaunig, H., and Rauner-Reithmayer, D. 2001. Goal-driven combination of software comprehension approaches for component based development. In Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (Toronto, Ontario, Canada). SSR '01. ACM Press, New York, NY, 95-102.	2		
[MIT02]	Roland T. Mittermeir and Heinz Pozewaunig. Selfdescriptive Software Components. In Proc. of the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002), Austria, April 2002.	2		
[MOR02]	Morisio, M. Torchiano, M. , "Definition and Classification of COTS: A Proposal", LECTURE NOTES IN COMPUTER SCIENCE, 2002, ISSU 2255, pages 165-175	24		
[MUE01]	Müller-Olm M , Seidl H, "On optimal slicing of parallel programs", Proceedings of the thirty-third annual ACM symposium on Theory of computing, Hersonissos, Greece July 2001, p.647-656.			
[MUL98]	H. Muller and K. Klashinsky, "Rigi – a system for programming-in-the-large". In Proceedings of the 10th International Conference on Software Engineering (ICSE 10), 1998	112		
[MUR95]	G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models". In Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995	192		
[MUR96]	G. Murphy, "Lightweight Structural Summarization as an Aid to Software Evolution". PhD Thesis, University of Washington, 1996	30		
[MUR96]	G. Murphy and D. Notkin. "Lightweight Lexical Source Model Extraction." ACM Transactions on Software Engineering and Methodology. Vol. 5, No. 3, pages 262-292, July, 1996.	116		
[NAV04]	Navicon Company, http://www.navicon.de			tool web page
[NEN00]	Lilli Nenonen, Juha Gustafsson, Jukka Paakki, A. Inkeri Verkamo, Measuring object-oriented software architectures from UML diagrams; In: Proceedings of the 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Sophia Antipolis, France, June 2000, 87-100.	3		
[OBE97]	T.Oberndorf. "COTS and Open Systems - An Overview". 1997, available at http://www.sei.cmu.edu/str/descriptions/cots.html#ndi			
[OBR02]	Liam O'Brien, Christoph Stoermer, Chris Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", Technical Report, CMU/SEI-2002-TR-024	12		
[ORS03]	A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," In proc. of the 9th ESEC and 10th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, 2003, pp 128-137.	29		
[PAG80]	M. Page-Jones. The Practical Guide to Structured Systems Design. YOURDON Press, New York, NY, 1980.			Book
[PAG91]	F. G. Pagan, "Partial Computation and the Construction of Language Processors". Prentice Hall, 1991			BOOK
[PAU01]	Raymond Paul, "End-to-End Integration Testing," 2nd Asia-Pacific Conference on Quality Software (APAQS), 2001, pp. 211-222.	17		IEEE
[PAU94]	S. Paul and A. Prakash, "A Framework for Source Code Search Using Program Patterns". IEEE Transactions of Software Engineering, 20(6), June 1994	120		
[PER98]	Perens, B. (1998). The Open Source Definition. http://www. opensource.org/docs/definition_plain.html. accessed 20060215.			
[PIN03]	Pinzger, M., Oberleitner, J., and Gall, H. 2003. Analyzing and Understanding Architectural Characteristics of COM+ Components. In Proceedings of the 11th IEEE international Workshop on Program Comprehension (May 10 - 11, 2003). IWPC. IEEE Computer Society, Washington, DC, 54.	6		
[QUE94]	J-P. Queille, J-F. Voidrot , M. Munro, N. Wilde, "The Impact Analysis Task in Software Maintenance: A Model and a Case Study," Int'l Conference on Software Maintenance, 1994.	26		IEEE
[RADZI]	Eimutis S. Radzius, "A Methodology for the Planning of a Scenario Based Test Program,"	3		IEEE

	http://www.usecasemaps.org/pub/test_planning_SBT.pdf			
[RAM01]	Ramesh, B. Jarke, M., Toward Reference Models of Requirements Tracebility, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2001, VOL 27; PART 1, pages 58-93	171		
[REI04]	REIFER Donald J.; BASILI Victor R. ; BOEHM Barry W.; CLARK Betsy; Kazman Rick ; Port Dan ;COTS-Based Systems – Twelve Lessons Learned about Maintenance	7		
[REI95]	Reiss, S.P., "An Engine for the 3D Visualization of Program Information",JOURNAL OF VISUAL LANGUAGES AND COMPUTING, 995, VOL 6; NUMBER 3, pages 299			
[REP97]	Reps, T.; Siff, M.; Identifying Modules via Concept Analysis, In Proceedings of the International Conference on Software Maintenance, Bari, Italy, Oct 1997, p.170-179.	127		
[RIC90]	C. Rich and R. Waters, "The Programmer's Apprentice". Addison-Wesley, Baltimore, Maryland, 1990	166		BOOK
[RIL03]	J. Rilling, State of the Art Report: System Architecture Recovery and Comprehension, Tech. Report, DRDC Valcartier, Val-Bélair, Que., 2003.			
[RIV04]	Claudio Riva. View-Based Software Architecture, Reconstruction. Ph.D. thesis, Vienna University of Technology, 2004.	6		
[ROU05]	Atanas Rountev, Component-Level Dataflow Analysis, Lecture Notes in Computer Science, Volume 3489, Jan 2005, Pages 82 - 89, DOI 10.1007/11424529_6, URL http://dx.doi.org/10.1007/11424529_6	2		
[SAL06]	Salah, M. Mancoridis, S. Antoniol, G. Di Penta, M. , "Scenario-driven dynamic analysis for comprehending large software systems",Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference Publication Date: 22-24 March 2006, Volume: 00, On page(s): 10 pp.-	2		
[SAR01]	Kamran Sartipi., Alborz: A Query-based Tool for Software Architecture Recovery. Proceedings of the IEEE International Workshop on Program Comprehension (IWPC 2001), Pages: 115-116, May 12-14, 2001, Toronto, Canada.	9		
[SCH91]	R. W. Schwanke, “An intelligent tool for re-engineering software modularity”, International Conference on Software Engineering, pages 83–92, May 1991.	148		IEEE
[SEI06}	http://www.sei.cmu.edu/architecture/			
[SEL94]	P. Selfridge and G. Heineman. "Graphical support for code-level software understanding". In Douglas Smith, editor, Proc. Of the 9th Conf. on Knowledge-Based Software Engineering (KBSE'94). IEEE Computer Society Press, 1994.	8		IEEE
[SET04]	Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, Chris DePalma, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts," 7th International Workshop on Principles of Software Evolution (IWPSE),	6		IEEE

	2004, pp. 49-54.			
[SNE00]	Snelting, G.; Tip, Frank; Reengineering Class Hierarchies Using Concept Analysis, ACM Transactions on Programming Languages and Systems (TOPLAS), May 2000, p.540-582.	117		
[SNE96]	Snelting, G.; Re-engineering of Configurations Based on Mathematical Concept Analysis, TOSEM 5(2), 1996.	81		
[SPU01]	T Souder, S Mancoridis, M Salah, "Form: a framework for creating views of program executions", Software Maintenance, 2001. Proceedings. IEEE International Conference on Publication Date: 2001 On page(s): 612-620	4		
[STR95]	M.-A. D. Storey and H. A. M uller, Manipulating and documenting software structures using shrimp views. To appear in Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95), Opio (Nice), France, October 16-20, 1995.			
[STR96]	M.R. Strens, R.C. Sugden, "Change Analysis: A Step towards Meeting the Challenge of Changing Requirements," IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), 1996, p. 278.	12		IEEE
[SZY98]	Szyperski, C. (1998) "Component Software: Beyond Object-Oriented Programming" Addison	9		
[TIG97]	Eirik Tryggeseth. Support for Understanding in Software Maintenance. PhD thesis, IDT, NTNU, October 1996. 353 p. (forthcoming PhD thesis)	2		
[TIL03]	Thomas Tilley, Richard Cole, Peter Becker, Peter Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities, Lecture Notes in Computer Science, Volume 3626, Jul 2005, Pages 250 - 271, DOI 10.1007/11528784_13, URL http://dx.doi.org/10.1007/11528784_13	19		IEEE
[TIP93]	Tip F., “A survey of program slicing techniques”, Journal of Progr. Languages, 3(3), pp. 121-189, 9/1995.			
[TON01]	Tonella, P.; Concept Analysis for Module Restructuring, Software Engineering, IEEE Transactions on , Volume 27 , Issue 4 , Apr 2001, p.351 - 363.	37		
[TON03]	Paolo Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," IEEE Trans. Software Engineering, 2003, 29(6), pp. 495-509.	15		
[TON03]	Tonella, P., Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis, IEEE Transactions on Software Engineering, Volume 29, No.6, Jun 2003.	15		
[TON04]	Paolo Tonella, Formal Concept Analysis in Software Engineering	1		
[TOS01]	ToscanaJ Project, DSTC, The University of Queensland, the Technical University of Darmstadt, http://toscanaj.sourceforge.net/			tool web page
[TSA01]	Wei-Tek Tsai, Xiaoying Bai, Ray J. Paul, Weiguang Shao, Vishal Agarwal, "End-To-End Integration Testing Design," 25th International Computer Software and Applications Conference (COMPSAC), 2001, pp. 166-	19		IEEE

	171.			
[TSA03]	W. T. Tsai, L. Yu, X. X. Liu, A. Said, Y. Xiao, "Scenario-Based Test Case Generation for State-Based Embedded Systems," http://asusrl.eas.asu.edu/Publications/IPCCC2003.pdf	10		IEEE
[TUR94]	R.J. Turver, M. Munro, An Early impact analysis technique for software maintenance, Journal of Software Maintenance: Research and Practice, Vol. 6 (1), 1994, pp. 35-52.	31		IEEE
[VIG96]	M.Vigder, M.Gentleman, J.Dean. "COTS Software Integration: State of the Art".Technical Report NRC No. 39190, 1996. (http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-39198_e.html)	38		
[VIG97]	Vigder, M.R., Dean, J. , An Architectural Approach to Building Systems from COTS Software Components, Proceedings of CASCON '97. Toronto, Ontario, Canada. November 10-13, 1997. pp. 131-143.	38		
[WAL02]	Kurt Wallnau, Scott Hissam, and Robert Seacord., Building Systems from Commercial Components, SEI 2002, ISBN: 0-201-70064-6			BOOK
[WAL05]	Walkinshaw, N., Roper, M., and Wood, M. 2005. Understanding Object-Oriented Source Code from the Behavioural Perspective. In Proceedings of the 13th international Workshop on Program Comprehension - Volume 00 (May 15 - 16, 2005). IWPC. IEEE Computer Society, Washington, DC, 215-224. DOI= http://dx.doi.org/10.1109/WPC.2005.44	2		
[WAN96]	Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, Sanjai Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis," 8th International Conference on Software Engineering and Knowledge Engineering (SEKE), 1996, pp.369-376.	14		
[WAT99]	Waters, R.; Rugaber, S.; Abowd, G.D.; Architectural Element Matching using Concept Analysis, Automated Software Engineering, 14th IEEE International Conference, Oct 1999, p. 291 -294.	2		
[WEI81]	M. Weiser. “Program Slicing”. Proceedings of the 5th International Conference on Software Engineering, San Diego, California, March 1981+B125	1291		hard copy only
[WEI98]	K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, "Scenarios in System Development: A Report on Current Practice," IEEE Software, 1998.	68		IEEE
[WEL97]	C. Welty, “Augmenting Abstract Syntax Trees for Program Understanding”. Proceedings of The 1997 International Conference on Automated Software Engineering. IEEE Computer Society Press. P. 126-133. November, 1997; http://www.cs.vassar.edu/faculty/welty/papers/ase-97_1.html	7		http://www.cs.vassar.edu/faculty/welty/papers/ase-97_1.html
[WIL94]	Queille, J.P., Voidrot, J.F., Wilde, N., Munro, M.; The Impact Analysis Task in Software Maintenance: A Model and a Case Study, In Proceedings of the International Conference on Software Maintenance, Victoria, Canada, IEEE Comp.Soc., Press, Sep 1994.	26		IEEE

[WIL95]	Wilde, N. Scully, M. C., "Software Reconnaissance: Mapping Program Features to Code",JOURNAL OF SOFTWARE MAINTENANCE 1995, VOL 7; NUMBER 1, pages 49	112		
[WU_92]	S. Wu and U. Manber, "Agrep-A Fast Approximately Pattern-Matching Tool". USENIX Winter 1992 Technical Conference, pages 153-162, San Francisco, U.S.A., 1992.	117		
[WUH04]	Jingwei Wu, Richard C. Holt, Ahmed E. Hassan, "Exploring Software Evolution Using Spectrographs," wcre, pp. 80-89, 11th Working Conference on Reverse Engineering (WCRE'04), 2004.	14		
[YEH97]	A.S. Yeh, D.R. Harris, M.P. Chase. “Manipulating Recovered Software Architecture Views.” Proceedings of the 19th International Conference on Software Engineering, pages 184-194, Boston, U.S.A., 1997.			
[ZHA02]	Jianjun Zhao, Hongji Yang, Liming Xiang, Baowen Xu, "Change Impact Analysis to Support Architectural Evolution," Journal of Software Maintenance 14(5), 317-333, 2002.	7		IEEE
[ZHA99]	Zhao J. ,” Slicing concurrent Java Programs”, Seventh International Workshop on Program Comprehension, May 05 - 07, 1999, Pittsburgh, Pennsylvania, pp 126-136.			

6.2 Sorted by Citations (descending order) – Threshole is 4+ citations

[WEI81]	M. Weiser. “Program Slicing”. Proceedings of the 5th International Conference on Software Engineering, San Diego, California, March 1981+B125	1291	
[JAI99]	Jain, A. K., Murty, M. N., and Flynn, P. J. 1999. Data clustering: a review. ACM Comput. Surv. 31, 3 (Sep. 1999), 264-323. DOI= http://doi.acm.org/10.1145/331499.331504	1153	
[BAA03]	F. Baader et al., "The Description Logic Handbook", Cambridge University Press, 2003	538	
[DBS91]	P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, “LaSSIE: a Knowledge-based Software Information System”. Communications of the ACM, 34(5):36–49, 1991	266	
[DEV91]	P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard, "LaSSIE: a Knowledge-based Software Information System". Communications of the ACM, 34(5):36-49, 1991	266	
[KAZ96]	Rick Kazman, Gregory D. Abowd, Leonard J. Bass, Paul C. Clements, "Scenario-Based Analysis of Software Architecture," IEEE Software 13(6), pp. 47-55 (1996)	235	
[KAZ96]	Kazman, Rick, 1995, Scenario-based analysis of software architecture: Waterloo, Ont.,Canada, University of Waterloo, Computer Science Dept..	235	
[KOR90]	B. Korel, J. Laski, “Dynamic slicing of computer programs,” Journal of Systems and Software, 13(3), pp.187-195, 1990.	233	
[DWY98]	Matthe w B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In Proceedings of the 21st Intl. Conf. on Software Engineering, 1999.	221	
[KOW97]	G. Kowalski, "Information Retrieval Systems: Theory and Implementation". Kluwer Academic Publishers, 1997	198	
[MUR95]	G. Murphy, D. Notkin, and K. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models”. In Proceedings of SIGSOFT’95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1995	192	
[CHE90]	Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, “The C Information Abstraction System”. IEEE Transactions on Software Engineering, 16(3):325-334, 1990	173	
[RAM01]	Ramesh, B. Jarke, M., Toward Reference Models of Requirements Tracebility, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2001, VOL 27; PART 1, pages 58-93	171	
[BIG94]	T. J. Biggerstaff, G. B. Mitbander, and D. Webster. “Program Understanding and the Concept Assignment Problem”. Communications of the ACM, 37(5):72-83, May 1994.	167	
[RIC90]	C. Rich and R. Waters, "The Programmer's Apprentice". Addison-Wesley, Baltimore, Maryland, 1990	166	
[FIN97]	Finnigan, P. J. Holt, R. C. Kalas, I. Kerr, S. Kontogiannis, K. Mueller, H. A. Mylopoulos, J. Perelgut, S. G. Stanley, M. Wong, K., The software bookshelf,IBM SYSTEMS JOURNAL, 1997, VOL 36; NUMBER 4, pages 564-593	163	
[SCH91]	R. W. Schwanke, “An intelligent tool for re-engineering software modularity”, International Conference on Software Engineering, pages 83–92, May 1991.	148	
[KAZ99]	Rick Kazman, S. Jeromy Carrière, Playing Detective: Reconstructing Software Architecture from Available Evidence, Automated Software	138	

	Engineering, Volume 6, Issue 2, Apr 1999, Pages 107 - 138		
[KRA96]	Kramer, C. and Prechelt, L. 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96) (November 08 - 10, 1996). WCRE. IEEE Computer Society, Washington, DC, 208.	128	
[BOH96]	S. Bohner and R. Arnold. Software Change Impact Analysis. IEEE Computer Society Press, 1996, pp. 1-26.	127	
[REP97]	Reps, T.; Siff, M.; Identifying Modules via Concept Analysis, In Proceedings of the International Conference on Software Maintenance, Bari, Italy, Oct 1997, p.170-179.	127	
[GAR01]	Marcela Genero, Jos Olivas, Mario Piattini, and Francisco Romero,2001,"Using metrics to predict OO information systems maintainability",13th International Conference Advanced Information Systems Engineering.	123	
[PAU94]	S. Paul and A. Prakash, “A Framework for Source Code Search Using Program Patterns”. IEEE Transactions of Software Engineering, 20(6), June 1994	120	
[LIN97]	Lindig, C.; Snelting, G.; Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, Software Engineering, 1997., In Proceedings of the 19th International Conference, May 1997, p.349-359.	119	
[SNE00]	Snelting, G.; Tip, Frank; Reengineering Class Hierarchies Using Concept Analysis, ACM Transactions on Programming Languages and Systems (TOPLAS), May 2000, p.540-582.	117	
[WU_92]	S. Wu and U. Manber, "Agrep-A Fast Approximately Pattern-Matching Tool". USENIX Winter 1992 Technical Conference, pages 153-162, San Francisco, U.S.A., 1992.	117	
[MUR96]	G. Murphy and D. Notkin. “Lightweight Lexical Source Model Extraction.” ACM Transactions on Software Engineering and Methodology. Vol. 5, No. 3, pages 262-292, July, 1996.	116	
[MUL98]	H. Muller and K. Klashinsky, “Rigi – a system for programming-in-the-large”. In Proceedings of the 10th International Conference on Software Engineering (ICSE 10), 1998	112	
[WIL95]	Wilde, N. Scully, M. C., "Software Reconnaissance: Mapping Program Features to Code",JOURNAL OF SOFTWARE MAINTENANCE 1995, VOL 7; NUMBER 1, pages 49	112	
[KUN94]	David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, Cris Chen, "Change Impact Identification in Object Oriented Software Maintenance," Int'l Conference on Software Maintenance,1994, pp. 202-211.	87	
[DEU99]	van Deursen, A.; Kuipers, T.; Identifying Objects Using Cluster and Concept Analysis, In Proceedings of the 1999 International Conference on, 16-22 May 1999, p. 246 -255.	85	
[SNE96]	Snelting, G.; Re-engineering of Configurations Based on Mathematical Concept Analysis, TOSEM 5(2), 1996.	81	
[BAS01]	Basili, V.R. and Boehm, B., "COTS-based systems top 10 list", Computer , 34(5), May 2001, pp. 91-95.	77	
[AHO79]	A. V. Aho, B. W. Kernighan, and P. J. Weinberger, "awk - A Pattern Scanning and Processing Language". Software Practice and Experience, Vol. 9, No. 9, pages 267-280, 1979	73	
[BRO00]	Lisa Brownsword, Tricia Oberndorf, Carol A. Sledge, "Developing New Processes for COTS-Based Systems," IEEE Software, vol. 17, no. 4, pp. 48-55, Jul/Aug, 2000.	70	

[KAZ98]	R. Kazman and S.J. Carrière. “View Extraction and View Fusion in Architectural Understanding.” Proceedings of the 5th International Conference of Software Reuse, Victoria, B.C., Canada, June, 1998.	70	
[WEI98]	K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, "Scenarios in System Development: A Report on Current Practice," IEEE Software, 1998.	68	
[BAL99]	Ball, Thomas, “The Concept of Dynamic Analysis”, ACM Conference on Foundations of Software Engineering, Toulouse,France, September 1999, pp.216-234.	66	
[ARN93]	R. S. Arnold, S. A. Bohner, "Impact Analysis - Towards a Framework for Comparison", Proc. Conf. Software Maintenan65ce, 1993, pp. 292-301.	65	
[FEI98]	L. Feijs, R. Krikhaar, and R. C. Ommering, “A Relational Approach to Support Software Architecture Analysis”. Software Practice and Experience, 28(4):371-400, 1998	55	
[KRI99]	Krikhaar, R. Software Architecture Reconstruction. Ph.D. Thesis, University of Amsterdam, June 1999.	49	
[LIN95]	Lindig, C.; Concept-based Component Retrieval, In Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors, Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs, Montreál, Aug 1995, p. 21-25.	46	
[CAN98]	G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, Information and Software Technology Special Issue on Program Slicing, volume 40, pages 595--607. Elsevier Science B. V., 1998.	42	
[AMY01]	D. Amyot, A. Eberlein, “An Evaluation of Scenario Notations for Telecommunication Systems Development,” 9th Int. Conference on Telecommunication Systems (9ICTS), Dallas, USA (March 2001).	40	
[VIG96]	M.Vigder, M.Gentleman, J.Dean. "COTS Software Integration: State of the Art".Technical Report NRC No. 39190, 1996. (http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-39198_e.html)	38	
[VIG97]	Vigder, M.R., Dean, J. , An Architectural Approach to Building Systems from COTS Software Components, Proceedings of CASCON '97. Toronto, Ontario, Canada. November 10-13, 1997. pp. 131-143.	38	
[HAR97]	Mark Harman, Sebastian Danicic"Amorphous Program Slicing",Proceedings of the 5th International Workshop on Program Comprehension (WPC '97), 1997,pp70	37	
[TON01]	Tonella, P.; Concept Analysis for Module Restructuring, Software Engineering, IEEE Transactions on , Volume 27 , Issue 4 , Apr 2001, p.351 - 363.	37	
[KOR99]	Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In Proc. Working Conference on Reverse Engineering, 1999	36	
[TUR94]	R.J. Turver, M. Munro, An Early impact analysis technique for software maintenance, Journal of Software Maintenance: Research and Practice, Vol. 6 (1), 1994, pp. 35-52.	31	
[MUR96]	G. Murphy, “Lightweight Structural Summarization as an Aid to Software Evolution”. PhD Thesis, University of Washington, 1996	30	
[ORS03]	A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging Field Data for Impact Analysis and Regression Testing," In proc. of the 9th ESEC and 10th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering, 2003, pp 128-137.	29	
[KUI00]	Kuipers, T.; Moonen, L.; Types and Concept Analysis for Legacy Systems, Program Comprehension, 2000, In Proceedings of the 8th International IWPC, Oct-Nov 2000.	27	

[KUL99]	B. Kullbach, and A. Winter, “Querying as an Enabling Technology in Software Reengineering”. In Proceedings of the 3rd EuroMicro Conference on Software Maintenance and Reengineering, 1999	27	
[EIS01]	Thomas Eisenbarth, Rainer Koschke, Daniel Simon, "Feature-Driven Program Understanding Using Concept Analysis of Execution Traces," iwpc, p. 0300, Ninth International Workshop on Program Comprehension (IWPC'01), 2001	26	
[HOL96]	R. C. Holt, “Binary relational algebra applied to software architecture”. Technical report, CSRI 345, University of Toronto, march 1996	26	
[QUE94]	J-P. Queille, J-F. Voidrot , M. Munro, N. Wilde, “The Impact Analysis Task in Software Maintenance: A Model and a Case Study,” Int’l Conference on Software Maintenance, 1994.	26	
[WIL94]	Queille, J.P., Voidrot, J.F., Wilde, N., Munro, M.; The Impact Analysis Task in Software Maintenance: A Model and a Case Study, In Proceedings of the International Conference on Software Maintenance, Victoria, Canada, IEEE Comp.Soc., Press, Sep 1994.	26	
[AND98]	J. Andrews, "Testing using Log File Analysis: Tools, Methods, and Issues," ase, p. 157, 13th IEEE International Conference on Automated Software Engineering (ASE'98), 1998	25	
[BER94]	A. Bergstra and P. Klint. The ToolBus -- a Component Interconnection Architecture--. P9408, Programming Research Group, University of Amsterdam, Amsterdam, 1994.	25	
[GRA92]	J. E. Grass, "Object-Oriented Design Archaeology with CIA++". Computing Systems, Journal of the USENIX Association, 5(1):5-67, Winter, 1992	24	
[KRI99]	R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, C. Verhoef, "A Two-Phase Process for Software Architecture Improvement," icsm, p. 371, 15th IEEE International Conference on Software Maintenance (ICSM'99), 1999.	24	
[MOR02]	Morisio, M. Torchiano, M. , “Definition and Classification of COTS: A Proposal”, LECTURE NOTES IN COMPUTER SCIENCE, 2002, ISSU 2255, pages 165-175	24	
[LAS02]	assing, N., Bengtsson, P., van Vliet, H., and Bosch, J. 2002. Experiences with ALMA: architecture-level modifiability analysis. J. Syst. Softw. 61, 1 (Mar. 2002), 47-57. DOI= http://dx.doi.org/10.1016/S0164-1212(01)00113-3	22	
[ANT00]	G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation". In Proceedings of IEEE International Conference on Software Maintenance, San Jose, CA, 2000	21	
[LAW03]	James Law, Gregg Rothermel, Whole Program Path-Based Dynamic Impact Analysis,” Internationall Conference on Software Engineering, 2003, pp. 308-318.	19	
[TIL03]	Thomas Tilley, Richard Cole, Peter Becker, Peter Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities, Lecture Notes in Computer Science, Volume 3626, Jul 2005, Pages 250 - 271, DOI 10.1007/11528784_13, URL http://dx.doi.org/10.1007/11528784_13	19	
[TSA01]	Wei-Tek Tsai, Xiaoying Bai, Ray J. Paul, Weiguang Shao, Vishal Agarwal, "End-To-End Integration Testing Design," 25th International Computer Software and Applications Conference (COMPSAC), 2001, pp. 166-171.	19	
[CAR99]	S. J. Cartiere, S. Woods, and R. Kazman, "Software architectural	18	

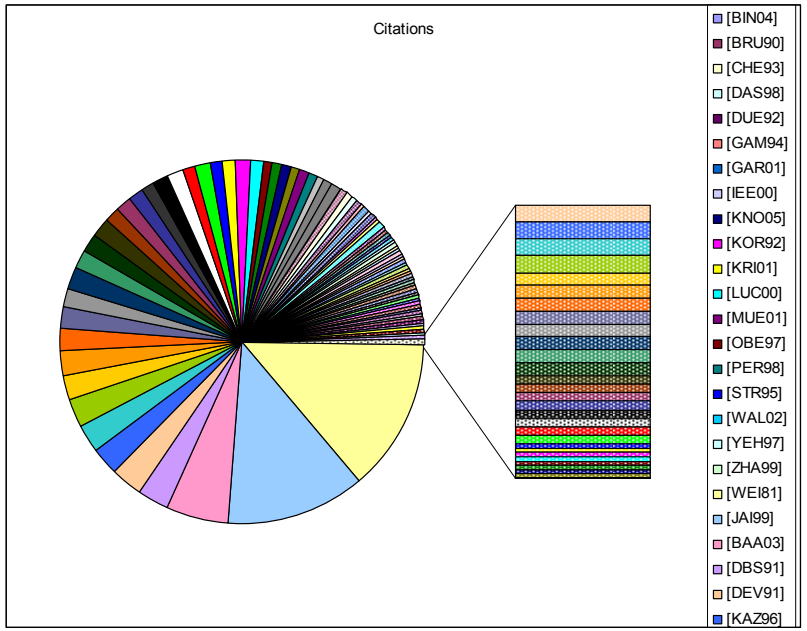
	transformation," in Proc. 6th Working Conf. Reverse Engineering, Oct. 1999.		
[KOS97]	Jean-Francois Girard, Rainer Koschke. A Comparison of Abstract Data Type and Objects Recovery Techniques. Journal Science of Computer Programming, Volume 36, Issue 2-3, pp. 149-181, Elsevier, March 2000	18	
[LEE00]	M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software," Computer Performance Evaluation: Modelling Techniques and Tools, 11th International Conference, TOOLS, 2000.	18	
[BER97]	Bergeron, J., Debbabi, M., Erhioui, M.M., and Ktari, B. 1999. Static analysis of binary code to isolate malicious behaviors, In Proc. IEEE International Workshop on Enterprise Security.	17	
[PAU01]	Raymond Paul, "End-to-End Integration Testing," 2nd Asia-Pacific Conference on Quality Software (APAQS), 2001, pp. 211-222.	17	
[FER01]	Ferenc, R., Magyar, F., Beszédes, Á., Kiss, Á. and Tarkiainen, M. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001), pages 16-27. Szeged, Hungary, June 15-16, 2001. Published by University of Szeged.	16	
[DEM99]	Serge Demeyer, Stphr, Ducasse and SanderTichPLLI("A Pattern Language for Reverse Engineering," Proceedings of th4th European Conference on Pattern Languages of Programming and Computing, 1999, Paul Dyson (Ed.), UVK Universittsverlag Konstanz GmbH, Konstanz, Germany, July 1999.	15	
[DEU05]	A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04)	15	
[GOL02]	N. Gold, K. Bennett, "Hypothesis-based Concept Assignment in Software Maintenance", IEEE Proceedings of Software Engineering, Vol 149, Issue 4, 2002	15	
[MEM03]	Atif Memon , Ishan Banerjee , Adithya Nagarajan, GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, Proceedings of the 10th Working Conference on Reverse Engineering, p.260, November 13-17, 2003	15	
[MET02]	Mehta, A. and Heineman, G. T. 2002. Evolving legacy system features into fine-grained components. In Proceedings of the 24th international Conference on Software Engineering (Orlando, Florida, May 19 - 25, 2002). ICSE '02. ACM Press, New York, NY, 417-427. DOI= http://doi.acm.org/10.1145/581339.581391	15	
[TON03]	Paolo Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," IEEE Trans. Software Engineering, 2003, 29(6), pp. 495-509.	15	
[TON03]	Tonella, P., Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis, IEEE Transactions on Software Engineering, Volume 29, No.6, Jun 2003.	15	
[KOS99]	Koschke, R. 1999. An Incremental Semi-Automatic Method for Component Recovery. In Proceedings of the Sixth Working Conference on Reverse Engineering (October 06 - 08, 1999). WCRE. IEEE Computer Society, Washington, DC, 256.	14	
[LAI83]	P. N. Johnson-Laird, "Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness". Harvard University,	14	

	Cambridge, Mass., 1983.		
[WAN96]	Yamin Wang, Wei-Tek Tsai, Xiaoping Chen, Sanjai Rayadurgam, "The Role of Program Slicing in Ripple Effect Analysis," 8th International Conference on Software Engineering and Knowledge Engineering (SEKE), 1996, pp.369-376.	14	
[WUH04]	Jingwei Wu, Richard C. Holt, Ahmed E. Hassan, "Exploring Software Evolution Using Spectrographs," wcre, pp. 80-89, 11th Working Conference on Reverse Engineering (WCRE'04), 2004.	14	
[BAI02]	Xiaoying Bai, Wei-Tek Tsai, Ke Feng, Lian Yu, Ray J. Paul, "Scenario-Based Modeling and Its Applications," WORDS 2002: 253-260.	13	
[CAN99]	Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A. (1999a), 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', Workshop on Program Comprehension, pp. 136-143, Pittsburgh, IEEE Computer Society Press.	13	
[CLA98]	J Clapp, A Taub - MP 98B0000069, " A Management Guide to Software Maintenance in COTS-Based Systems", The MITRE Corporation, Bedford, MA, November, 1998 - mitre.org	12	
[HOG00]	Kathi Hoggshead Davis and Peter Aiken "Data Reverse Engineering: An Historical Survey" Proceedings of the 7th Working Conference on Reverse Engineering Brisbane, Queensland, Australia • November 23-25, 2000 • pages 70-78	12	
[OBR02]	Liam O'Brien, Christoph Stoermer, Chris Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", Technical Report, CMU/SEI-2002-TR-024	12	
[STR96]	M.R. Strens, R.C. Sugden, "Change Analysis: A Step towards Meeting the Challenge of Changing Requirements," IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS'96), 1996, p. 278.	12	
[BUL02]	R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey, "Semantic Grep: Regular Expression + Relational Abstraction", In Proc of the 9th Working Conference on Reverse Engineering (WCRE'02), 2002	10	
[CIF97]	Cristina Cifuentes and Antoine Fraboulet, "Intraprocedural Static Slicing of Binary Executables", ICSM 199, pp.180	10	
[KLI03]	P. Klint, "How Understanding and Restructuring Differ from Compiling – A Rewriting Perspective". 11th IEEE International Workshop on Program Comprehension, 2003.	10	
[TSA03]	W. T. Tsai, L. Yu, X. X. Liu, A. Said, Y. Xiao, "Scenario-Based Test Case Generation for State-Based Embedded Systems," http://asusrl.eas.asu.edu/Publications/IPCCC2003.pdf	10	
[AND02]	A. Andrews, S. Ghosh, and E. M. Choi. A model for understanding software components. In Proc. of the International Conference on Software Maintenance, pages 359–368, Montreal, Canada, October 2002. IEEE Computer Society Press\	9	
[BRE00]	K. Breitman, J. C. Sampaio do Prado, "Scenario Evolution: A Closer View on Relationships," Proceedings. 4th International Conference on Requirements Engineering, 2000, pp. 95 - 105.	9	
[BRE00]	K. Breitman, J. C. Sampaio do Prado, "Scenario Evolution: A Closer View on Relationships," Proceedings. 4th International Conference on Requirements Engineering, 2000, pp. 95 - 105.	9	
[SAR01]	Kamran Sartipi., Alborz: A Query-based Tool for Software Architecture Recovery. Proceedings of the IEEE International Workshop on Program Comprehension (IWPC 2001), Pages: 115-116, May 12-14, 2001, Toronto, Canada.	9	

[SZY98]	Szyperski, C. (1998) "Component Software: Beyond Object-Oriented Programming" Addison	9	
[HAD02]	Haddox, J. M. Michael, C. C. Kapfhammer, G. M., "AN APPROACH FOR UNDERSTANDING AND TESTING THIRD PARTY SOFTWARE COMPONENTS", IEEE PROCEEDINGS OF THE ANNUAL RELIABILITY, AND MAINTAINABILITY SYMPOSIUM, 2002, ISSU 2002, pages 293-299	8	
[KVA05]	Kvale, A. A., Li, J., and Conradi, R. 2005. A case study on building COTS-based system using aspect-oriented programming. In Proceedings of the 2005 ACM Symposium on Applied Computing (Santa Fe, New Mexico, March 13 - 17, 2005). L. M. Liebrock, Ed. SAC '05. ACM Press, New York, NY, 1491-1498.	8	
[LAN01]	C. Lange, H. M. Sneed, and A. Winter, “Applying the graph-oriented GUPRO Approach in comparison to a Relational Database based Approach”. In Proc of the 9th International Workshop on Program Comprehension, 2001	8	
[MAR04]	A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code". in the Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, pp. 214-223, November 9-12, 2004	8	
[SEL94]	P. Selfridge and G. Heineman. "Graphical support for code-level software understanding". In Douglas Smith, editor, Proc. Of the 9th Conf. on Knowledge-Based Software Engineering (KBSE'94). IEEE Computer Society Press, 1994.	8	
[REI04]	REIFER Donald J.; BASILI Victor R. ; BOEHM Barry W.; CLARK Betsy; Kazman Rick ; Port Dan ;COTS-Based Systems – Twelve Lessons Learned about Maintenance	7	
[WEL97]	C. Welty, “Augmenting Abstract Syntax Trees for Program Understanding”. Proceedings of The 1997 International Conference on Automated Software Engineering. IEEE Computer Society Press. P. 126-133. November, 1997; http://www.cs.vassar.edu/faculty/welty/papers/ase-97_1.html	7	
[ZHA02]	Jianjun Zhao, Hongji Yang, Liming Xiang, Baowen Xu, "Change Impact Analysis to Support Architectural Evolution," Journal of Software Maintenance 14(5), 317-333, 2002.	7	
[BLA01]	Sue Black, "Computing Ripple Effect for Software Maintenance," Journal of Software Maintenance 13(4), pp. 263-279, 2001.	6	
[KIS05]	Ákos Kiss, Judit Jász, Tibor Gyimóthy, Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables, Software Quality Journal, Volume 13, Issue 3, Sep 2005, Pages 227 - 245,	6	
[PIN03]	Pinzger, M., Oberleitner, J., and Gall, H. 2003. Analyzing and Understanding Architectural Characteristics of COM+ Components. In Proceedings of the 11th IEEE international Workshop on Program Comprehension (May 10 - 11, 2003). IWPC. IEEE Computer Society, Washington, DC, 54.	6	
[RIV04]	Claudio Riva. View-Based Software Architecture, Reconstruction. Ph.D. thesis, Vienna University of Technology, 2004.	6	
[SET04]	Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, Chris DePalma, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts," 7th International Workshop on Principles of Software Evolution (IWPSE), 2004, pp. 49-54.	6	
[MCK04]	P.K. McKinley et al., “A Taxonomy of Compositional Adaptation,” tech. report MSU-CSE-04-17, Dept. Computer Science and	5	

	Engineering, Michigan State Univ., 2004		
[BRU93]	B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analysis. In Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSLA93), Washington, USA, September 1993.	4	
[DEU04]	A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-Driven Software Architecture Reconstruction. In Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WICSA'04).	4	
[HAM01]	Hamou-Lhadj, Abdelwahab, and Timothy C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, Markham, Canada, October 2004, pp.42-55.	4	
[KOS02]	Rainer Koschke und Yan Zhang, Component Recovery, Protocol Recovery and Validation. In: 3. Workshop Software-Reengineering, Bad Honnef (10./11.Mai 2001), Fachberichte Informatik, Universität Koblenz-Landau, Nr. 1/2002, pages 73-76, Januar 2002.	4	
[SPU01]	T Souder, S Mancoridis, M Salah, "Form: a framework for creating views of program executions", Software Maintenance, 2001. Proceedings. IEEE International Conference on Publication Date: 2001 On page(s): 612-620	4	

6.3 Citation Index (Pie chart)



6.4 Citations ranking (bar chart)