

Reverse-Engineering and Configuration Management: Concepts and perspectives

ADELE team
IMAG Institute - CNRS
University of Grenoble I
Laboratoire LSR, Actimart Bat 8, Av. de vignate, 38610 Gières
Tel. (33) 76 63 34 66 Fax. (33) 76 63 34 58
jmfavre@imag.fr

Abstract

More than one decade of industrial experience with the Adele CM environment shows that advances in software configuration management are effective only if forward-looking research can be merged with pragmatic work on present and past technology. Based on our experience in the CM field and starting from the observation that the technology today in use was designed in the 70's, this paper tries to answer the following questions: How can we support the maintenance of existing configuration management artifacts? What are their relationships with other software artifacts? Is it possible to facilitate the adoption of modern configuration tools? What about the intersection between reverse engineering and configuration management?

Keywords

Software maintenance, Configuration Management, Reverse engineering, Reengineering, Programming-in-the-large

Cursus

CNRS. Ph.D. University of Grenoble I. Working in the ADELE team since 1987 under the direction of Jacky Estublier. Research focusing on the concepts and techniques useful to maintain large existing software. Special focus on software maintenance, configuration management and reverse engineering. Lecturer at the UCV (University Central of Venezuela) during two years, giving lectures on software engineering, maintenance and configuration management.

1 Programming-in-the-large

The large range of functionalities covered by modern CM-related tools leads to great difficulties in defining the "configuration management" term precisely. Indeed, it is not clear what the dividing line is between what is CM and what is not CM [Dar92]. When more precision is needed, it is more convenient to use a *conceptual* terminology rather than a terminology driven by hazardous *technological* progress.

1.1 Programming-in-the-large classification

Programming-in-the-small (PITS) is concerned with two fundamental notions: **algorithms** and **data-structures**. It corresponds to "traditional" programming activities. Attention is devoted to the details of individual software components. Typical entities of interest are variables, expressions, statement, etc.

Programming-in-the-large (PITL) focuses on concepts related with the management of large software products, taking into account four aspects: architecture, manufacture, evolution and variation.

- **Architecture** (or structure). The software architecture defines complex software products as a composition of numerous components. The **Architecture-in-the-small** (or detailed architecture) field corresponds to the control of the interactions between programming language entities, such as procedures or global variables. On the contrary the **Architecture-in-the-large** (or global architecture) field is related with the composition of systems and subsystems starting from these modules.
- **Manufacture** (or building). The distinction is made

between source objects and derived objects.

- **Evolution**: Software evolution cannot be avoided. This phenomenon is intrinsically linked to the time notion. The point of interest here is not the evolution process in itself, but the *result* of this process on the software product. Revisions and historical information (e.g. evolution change requests, release notes, etc.) are basic entities.
- **Variation**: The success of software often depends on its ability to operate on various platforms and to accommodate a large variety of users and organizations. When a single version of the product does not cover all needs, different variants have to be implemented. An appropriate grain level should be selected. The **Variation-in-the-small** (or detailed variation) term is used when all software variants are represented into a single file (conditional compilation is a typical implementation technique). By contrast, **Variation-in-the-large** (or global variation) corresponds to the case where each variant is represented as a separate file. In the first case, information-sharing is privileged; in the second case information is duplicated but the overall structure is usually easier to understand.

Note that variation is not necessarily connected with the evolution concept. The existence of different variants are independent from the time notion: the need for them can have been identified from the start of the software project.

Programming-In-The-Many (PITM) concepts are related to the presence of multiple (human) agents cooperating in the software development process. Basic notions are resources (e.g. staff, time, budget, hardware, etc.) and activities. While PITS and PITL concepts are relative to the *software product*,

PITM corresponds to the *software process*.

Obviously the classification presented above is an over simplification. Connections exist at the conceptual level. For instance a configuration is an assembly of multiple components (architecture) existing in different variants (variation) and/or revisions (evolution). Connections (and sometimes confusions) are also due to technological features. In almost all the cases, the functionalities provided by a single tool cross the limits presented above. Even a simple tool like SCCS addresses both PITL problems (via the representation of revisions) and PITM problems (via the built-in check-in/check-out process). Indeed, most configuration management tools cover functionalities related to different fields, ranging from PITL to PITM.

Finally, “**configuration management**” should be seen as a rallying term including both PITL and PITM concepts. In this paper, attention is focused on its impact on software products (the PITL part).

1.2 PITL state-of-the-art

Nowadays, a wide variety of tools address some PITL functionalities. These range from programming languages, individual tools covering only few functionalities, to complex PITL environments addressing a large subset of functionalities (software repositories, configuration management systems, etc.) Such environments also support some PITM concepts.

Architecture-in-the-small concepts have been included in modular languages like Ada or Modula 2. Some module interconnection languages (MIL) also describe such information, but in a separate artifact. Finally, modern PITL software environments make possible to represent architecture-in-the-large information in terms of powerful software product models. These models are typically based on an entity-relationship model extended with object-oriented facilities. They make possible to manage all software artifacts (programs, documentation, test cases, etc.) as typed information. Adele is provide such facilities.

State-of-the-art systems addressing manufacturing problems also take into account some aspects of software variation and evolution. This is necessary because the manufacturing process in itself must be adapted to different platforms, setting appropriate compilation options, while selecting revisions and variants of software components. Recent proposals include the Proteus configuration language [TG95], the Vesta system based on a polymorphic functional language and the Derive system based on a deductive database. Other environments put more emphasis on evolution and variation problems, sometimes providing uniform solutions (e.g. CoV [Mun93] or Ice [Zel94]).

The overall trend is to integrate in configuration management, concepts and/or technologies developed in other fields: object orientation; active, deductive and historical databases; feature logics, etc.

1.3 PITL state-of-the-practice

The state of the practice is less exciting... For most existing software, there is no explicit architecture-in-the-small description. Traditional programming languages like Fortran or C provide no specific support for PITL. Architecture-in-the-large is not represented explicitly either. In most cases, design documents are out-of-date and the only concrete representation of systems and subsystems is the hierarchical organization of source code in term of directories and sub-directories, as well as

the organization of code objects into libraries. Object types represented in the file system, must be deduced by special file name conventions (typically file name extensions). There is usually no explicit representation of inter-module or intersystem connections.

Software manufacture is usually automated by means of shell scripts, job control languages or make files. Make is probably one of the most popular software engineering tools in use today.

Software variants are represented at both grain levels. The variation-in-the-large solutions correspond to the cases where source codes or derived objects are stored in different directories, or where file name conventions are used for each variant. During the manufacturing process, the selection of source objects is classically realized through derivation tool's path options (e.g. -I or -L for Unix compilers). Otherwise, a distinct shell script is given for manufacturing each specific variant. Variation-in-the-small solutions are preferred when there are many similarities between variants. All variants are gathered into a single file, specific parts being included between conditional compilation directives. Preprocessors like CPP, are commonly used. In large software, variation constraints are implicitly represented in make files, include files, shell-script, etc.

Similar practices are used to represent the result of the software product evolution. For instance, file name conventions and multiple directory structures are common ways of keeping successive revisions of the system. When available, tools like SCCS, are also used to save disk space. Historical information about the software, like change request forms or modification reports are stored in text files, proprietary databases, or use a third party tools.

In conclusion, it is interesting to note that most of the tools cited above were designed before or during the 70's. For instance preprocessor technology has been popular since the 60's; CPP (and the C language) was defined in the second part of the 70's; Make has been in heavy use since 1975; SCCS in 1974, etc.

1.4 Programming-in-the-large problems

Clearly, there is a big step between the state-of-the-art and the state-of-the-practice.

1.4.1 *The Programming-in-the-large field is immature*

Some estimations suggest that less than 10% of programmers on the planet use a modern configuration manager [IBW93]. In fact, the maturity level of the PITL field is really low, especially when compared with PITS. Despite their importance in software production, PITL concepts are usually not included in traditional computer science curricula. Most managers feel that software engineers are trained to develop small systems, but not to deal with the evolution of large software products. PITL training is done on the job. This low level of education results in situations where most of the work is done empirically. PITL is not seen as a conceptual concern, but as a technical one. In small organizations, software engineers rely on services provided by the operating system. All the tools are used together in an ad hoc manner.

1.4.2 *Maintaining large software products is difficult*

Immaturity, poor education and poor tool support also result in software products that are difficult to understand and therefore to maintain. Architectural complexity is the prime factor leading to

difficulties. No matter how cleverly an architecture is designed early on in a project, it is likely to be changed later when technical obstacles are discovered, when software has to be adapted to a new platform or when new requirements have to be included. Without discipline and appropriate representation techniques, almost nobody has a good vision of the overall structure of the product. Without an overall understanding, changes are implemented in a blind way. This implies further destructuring, making the system even more complex to understand, and so on.

Software variation makes maintenance problems even more acute! When multiple variants are to be maintained, each change should be designed very carefully. On the one hand, variation-in-the-large implementation techniques lead to the so-called “multiple maintenance problem” [Mah94]: since information is duplicated, maintainers have to apply changes successively to all variants. On the other hand, while variation-in-the-small techniques like conditional compilation avoid this duplication of effort, interweaved variant structures can be virtually unreadable by humans [SC92] [Fav95b]. Constraints associated with activities like porting are usually very strict (unknown platform, remote porting site, short delivery times, etc.). Often, these constraints prevent doing the right thing in the first place.

In practice, all the problems presented above occur simultaneously. The maintainer's challenge is to deal with evolving multi-variant architectures and to gather all PITL informations needed to implement a change. Practical issues make this task difficult: while it is usually possible to get an idea of what an algorithm does by reading a few pages on the screen, PITL information is spread in multiple files among the file system and in some cases different formalisms are used. For instance, understanding the precise purpose of a textual inclusion directive in portable software, can require browsing many files, in many directories, on many platforms. Indeed, this examination process has to be done in an organized manner because the order in which preprocessor directives are processed, may be relevant. What is more, since the compilation options may change preprocessor behavior, make files and shell scripts should also be examined. Often, scanning all these files on different platforms is impossible or too painful. So maintainers are forced to make assumptions, and maintenance becomes a “hit or miss” process.

1.4.3 Modern CM tool adoption is a slow, difficult and expensive process

It is striking that PITL tools from the 70's are still in heavy use even if modern technology is now available. Indeed, the gap existing between the state-of-the-art and the state-of-the-practice is in agreement with studies suggesting that the length of time to transfer an innovative software idea into widespread adoption and use, is 15 to 20 years [Sca94]. In fact, most software engineers feel that tools like CPP or Make are rather convenient: even if such tools present evident drawbacks, at least they allow them to do their job. This trend is partly due to the general acceptance that PITL problems are technical ones and that their complexity justifies empirical solutions. Surveys in large maintenance organizations also show that maintenance teams ignore which are the modern CM tools available on the market, or at least what functionalities are provided [DCB93]. In fact, growing interest in configuration management is not led by development or maintenance teams, but mainly by management staff. As organizations begin to more formally define their process models and evaluate their process maturity level, e.g.,

based on the SEI's CMM process maturity levels, it becomes clear that CM is a key factor in attaining a higher process maturity level [Dart92]. Modern CM tool adoption is a slow, difficult and expensive process [Dar92][AT96].

Recently, the CM industry has become big business. Estimations suggest that the value of the market is increasing by 50% per year, to reach approximately \$1 billion in 1997, revenues from consultancy being included [IBW93]. More than 50 vendors are in competition, but no single system provides all the concepts to suit all the needs of customers. From the customer's point of view, selecting an appropriate CM tool is a difficult but critical task. One of the major difficulties for CM vendors, is to show immediate benefit from using their tool on the client's existing software products. Moreover, developers and maintainers are usually reluctant to use new and disrupting tools. Thus, the facility with which existing software are brought into the CM environment, can be a key factor in the CM selection process.

1.4.4 Integration between technologies should be improved

All PITS tools are based on file system technology. Since this technology will remain for a long time, PITL tools have to deal with it. That is to say, CM environment should provide, not only a way to import data from existing software products, but also features to manage multiple representations of the same entities in a consistent way. While a traditional view of the product is necessary for developers and maintainers working with existing traditional tools, other software engineering activities should benefit from powerful modelling of the same product. In other words, integration between old and new technology is necessary.

Integration is also needed between coarse and fine grain levels. PITL information is usually represented both with programming languages and PITL environments. Consistency should thus be ensured. The dependency relation between modules is a classical example. While represented explicitly in PITL environments, make files, or other PITL artifacts, it should be consistent with the relationships extractable from source codes.

1.4.5 Synthesis

Summing up, the following programming-in-the-large problems have been identified:

- Current PITL representations are not satisfactory. Existing software products are difficult to understand and therefore to maintain.
- There is a big divide between the state-of-the-art and the state-of-the-practice. Integration of existing software products in modern PITL environment is an important issue
- To be useful, practical PITL environments should integrate smoothly new and old technologies.

Quoting Dart, “the CM solution is as complex as its problem” [Dar92]. We believe that the only viable solution to prepare the future is to build on the past.

2 Reverse-engineering

As is the case for configuration management, reverse engineering popularity has increased significantly since the beginning of this decade, even if it was practiced before. The reverse engineering term, while being defined precisely, is also used as a generic term when it is not necessary to go into details.

2.1 Basic reversing engineering classification

Reverse engineering (RE) is “the process of analyzing a subject system with two goals in mind: (1) to identify the system’s components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction” [CC90]. **Redocumentation** is the creation or revision of alternate views semantically coherent with the examined system. This is a specialization of reverse engineering. Reverse engineering do not change the system observed.

By contrast, restructuring and reengineering do. “**Restructuring** is the transformation from one representation to another at the same relative abstraction level, while preserving the subject system’s external behavior”. This include among other, simplification of the representation, elimination of dead-code, adjustment to conform to new standard, etc. **Reengineering** is “the examination of a subject systems to reconstitute it in a new form and the subsequent implementation of the new form”. It usually involve a reverse-engineering phase followed by forward-engineering phase. Unlike restructuring, this process can change the semantic of the system. **Modernization** is a specialization of restructuring or reengineering (depending on whether or not the functionality change) to the case where the new representation is based on a technology newer that those used by the original representation.

This classification is a conceptual one. It does not refer to any specific technic, neither it indicates the degree of automation of related activities. Restructuring a program can be done using only a textual editor. Clearly, the key issue is to assist RE activities whenever possible.

2.2 Reverse engineering classification by objectives

The basic RE taxonomy does not specify any objective. Objectives can be used to form an orthogonal taxonomy.

RE is useful when the representation of a software is not jugged satisfactory, but also when not enough information is available about it. RE is not necessary associated with legacy systems and old technologies. The subject system can be either a new or old software product. In the former case, the objective is to generate alternate views to document the software. This can be useful, even if the representation is satisfactory. For convenience, let’s call this RE specialization **reverse-engineering-for-documenting**. When the subject system is an old one, the RE goal can be, (1) to facilitate the maintenance of this product or, (2) to reuse some of its components (or embedded “know-how”) to develop a new product. Let’s call these two RE specializations **reverse-engineering-for-maintenance** and **reverse-engineering-for-reuse**.

Note that, the fact that a product is judged satisfactory or not, depends on the current state of the product, but also on the evolution of quality criteria. For instance, software can be judged to be destructured, because of the effect of multiple changes on it, but also because state-of-the-art evolution has introduced a better way to structure programs. Indeed, over the years, RE objectives tend naturally to follow the state-of-the-art evolution. As an illustration, the shifts from ad hoc programming to structured programming, then to modular programming, and now to object-oriented programming led to respective interests in eliminating Goto statements, modularizing software systems, and now recovering objects from source code. In this sense, RE can be seen as a bridge between the state-of-the-art and the state-

of-the-practice.

Constraints on software representation are not only dictated by the state-of-the-art, but also by tools. Using a tool can imply adjusting the existing representation, extracting information from it, etc. More generally, when the goal of an RE activity is to bring a software product, or information about it, under the control of a tool, we will use the term **reverse-engineering-for-integration**. With the recent development of CASE technology, the importance of these activities is increasing. They should be assisted whenever possible. For CASE tool vendors, providing RE tools opens the markets of existing software. Indeed, when *old* software are integrated into *new* tools, RE can be seen as a kind of technology transfer vehicle.

3 Reverse-engineering-in-the-large

When programming-in-the-large problems are compared with reverse engineering objectives, the necessity to study the intersection between these two fields is evident.

3.1 Reverse-engineering-in-the-large classification

Neither of the two RE classification presented above, depends on the nature of artifacts being examined. Therefore, these definitions can be naturally combined with our PITL classification, giving rise to an another orthogonal classification.

Reverse-engineering-in-the-small (REITS) is the specialization of reverse engineering where the focus of attention is on programming-in-the-small concepts, namely algorithm and data structures.

Reverse-engineering-in-the-large (REITL) is the specialization of reverse engineering where the focus of attention is on programming-in-the-large concepts, namely architecture, manufacture, evolution and variation¹.

This last classification is orthogonal with the two others. We thus obtain a 3 dimensional space; each dimensions corresponds to a question: “what (is done)?”; “on what?”, “why?”. Figure XXX illustrates this breakdown. Even if it is not perfect (frontiers are not always well defined), it results in a convenient and systematic way to study the field. Each point can be referred to by its coordinates (“on what?”, “what?”, “why?”). For instance, we can consider “algorithm-reengineering-for-reuse”, “architecture-redocumentation-for-maintenance”, etc. This decomposition does not prevent using other dimensions when needed. For instance, it is sometimes useful to focus on techniques, i.e. on “how?” questions (visualization, animation, static analysis, specialization, query-language, etc.). This produce “algorithm-animation”, “evolution-visualization”, etc.

3.2 Reverse-engineering-in-the-small

Since PITS is more mature than PITL, it is not surprising that reverse engineering mainly focus on PITS. In order to make the

1. It is also possible to defined reverse-engineering-in-the-many (REITM). While the software product play the role of the subject for REITS and REITL activities, the software production process is analyzed and eventually changed in REITM activities. Reverse engineering is usually called “software process capture”, re-engineering correspond to “*software process improvement*”. These concepts should not be confused with the “*business process reengineering*” (BPR): the subject process to be improved is not the same and the software do not play the same role in each case.

classification more intuitive, the next tables illustrate its use with basic examples. “Why?” and “What?” dimensions are used.

RE--in-the-small (Why?)	examples
for-documentation	generating control and data flow diagrams.
for-maintenance.	eliminating of Goto statements to simplify the code structure.
for-reuse	examining Fortran libraries to integrate some algorithm into a new software.
for-integration	moving from C to ansi-C to use a PITS environment.

RE--in-the-small (What?)	(on what?) examples
Reverse-engineering	(1) algorithm. Extracting specifications form source code. (2) data. Generating ER-diagram from cobol programs.
Redocumentation	(1) algorithm. Generating control flow diagram. (2) data. Generating a data dictionary from cobol source code
Restructuring	(1) algorithm. Elimination of Goto statements. (2) data. Data name rationalization; data restructuring.
Reengineering	(1) algorithm. specifications extraction and re-implementation. (2) data. Re-implementation of complex data structures.
Modernization	(1) algorithm. Translation from Fortran to Ada. (2) data. from Cobol file structures to relational database.

Many techniques and tools have been developed to support RE-in-the-small, including algorithm-animation, data-structure-visualization, algorithm-pattern-recognition at different level (syntactic, semantic, plan, etc.), algorithm-slicing, etc.

3.3 Architecture-reverse-engineering

As mentioned above, the architecture quality is determinant for large software products. Programming languages represent implicitly architecture-in-the-small informations; file system hierarchies are used as ad-hoc architecture-in-the-large representations. Architecture-RE activities have been practiced for many years in industry, but without specific tool support. Important research efforts have been led in recent years. The next tables show different examples of architecture-RE activities.

Architecture-RE (Why?)	examples
for-documentation	Generating inter-module dependency graphs. (This allows, for instance, to compare expected and implemented architectures)
for-maintenance.	(Re)modularisation to facilitate future maintenance
for-reuse	Reusing architectural design within a same application domain
for-integration	Extracting inter-module dependencies from source-code to import a software product into a modern CM tool like Adele

Architecture RE (What?)	examples
Reverse-engineering	recovering module interconnection language information from source code
Redocumentation	generation of inter-modular dependency graphs. Generation of precise interfaces from source code.
Restructuring	(Re)modularisation (i.e. changing the module-procedure composition relation). elimination of unused procedures. clustering different modules into an new library; changing the system-subsystem relation, etc.
Modernization	changing an architecture-in-the-large file system representation into an entity-relationship data representation managed by a CM tool

Various technics and tools have been proposed (Cia from ATT, Rigi [Ma192], Arch from Siemens, Field, etc.). These include, architecture-visualization [BE94], interface-slicing [BC93], architecture-pattern-recognition, architecture-query-language [GC90] [CMR92], etc.

3.4 Manufacture-reverse-engineering

Manufacture-RE importance is dependent on the software. While being obviously not a critical issue compared to

Architecture-RE, it requires attention, especially in the case of large portable software products. In some cases, the manufacture description is based on hundreds of shell scripts or make files. Even if understanding and maintaining such complex structures is tedious, almost no tool support is provided. The following table describes common practices or research topics.

3.4.1 Why?

- **Manufacture-RE-for-documentation.** Generating derivation graphs or animating the manufacturing process can be useful, even when a new software is being developed. In large organization, the persons which write the manufacture description and which use it, are not always the same, but everybody should understand the manufacture process...
- **Manufacture-RE-for-maintenance.** Multi-platform manufacture descriptions are usually difficult to maintain and to port to new platform. Dead code should be eliminated, platform requirement should be redocumented (i.e. which tools are used, which libraries, etc.).
- **Manufacture-RE-for-reuse.** Manufacture description are usually reused in different project. A common practice is to get a make file from a project, to remove unused functionalities, to add rules for new derivation tools, etc. This process can be complex when shell scripts are used and various platforms supported.
- **Manufacture-RE-for-integration.** When the manufacture process is intended to be supported by a new tool, manufacture-RE activities take place, e.g. reengineering shell scripts to use Make, or adjusting make files to parallel make.

3.4.2 What?.

Manufacture RE (What?)	examples
Reverse-engineering	extracting derivation rules between software artifacts from shell scripts or job control language
Redocumentation	generating the derivation graph or extracting manufacture requirement w.r.t. to the platform
Restructuring	eliminating unused derivations; optimizing the manufacturing process by means of parallelisation or the introduction of intermediate steps. splitting makefiles, etc.
Modernization	assisting the conversion from make files to modern manufacture language such as Proteus

3.5 Evolution-reverse-engineering

At the first sight, Evolution-RE seems to be a non-sense. In fact, this paper focus on the software product, not the software process. Evolution-RE refers to the reverse engineering of the artifacts resulting from software evolution, namely product revisions and associated historical informations like change request formularies, comments in source code, etc. Clearly, since one can not change the past, maintaining these artifacts is meaningless¹. By contrast, examining the past may be useful in many situations. The following table briefly analyze this concept.

3.5.1 Why?

- **Evolution-RE-for-documentation.** Analyzing the evolution of a software is obviously important from a management point of view. This can be done both during development and maintenance.

1. Note that this is a fundamental difference between the evolution and the variation concepts: while variants must be maintained, this is not the case for revisions.

- **Evolution-RE-for-maintenance.** Maintainers often ask “What have been changed in the last versions?”. Facilitating the collection of such information is a typical task for CM tool. The problem is to present it in a useful way, to generate alternate views, ranging from to global view to detailed ones, etc. When many changes must be implemented in short delays, maintainers tend to implement them without providing appropriated historical informations. Documenting a major software release is usually done afterwards. This task, which involve the a posteriori examination of software evolution artifacts, is a typical Evolution-reverse-engineering activity.
- **Evolution-RE-for-reuse.** On possible objective in observing past evolution, is to reuse experiences from one project to another. This is an important issue for the software process improvement.
- **Evolution-RE-for-integration.** When the software product history is considered as important, restructuring (but not modification) of the corresponding artifacts can take place. For instance importing SCCS archives into modern CM environment is a possible task.

3.5.2 What?

- **Evolution-reverse-engineering.** One possible goal is to understand why a given change in the source code was done, that is to say, the problem is to recover change design information from change implementation.
- **Evolution-redocumentation.** While pretty printing is one of the weakest form of redocumentation-in-the-small, tools computing textual differences between files provided elementary support for evolution-redocumentation. Clearly other powerful visualization techniques are needed.

3.5.3 How?

Currently few existing techniques can support evolution-RE activities, but more work should be done, for instance for evolution-visualization, evolution-animation, evolution-query-language, etc. Ideally such techniques should be integrated in any modern CM tool environment.

3.6 Variation-reverse-engineering

Variation-reverse-engineering is the reverse engineering of software artifacts due to software variation (referred as software variants for simplicity). As mentioned in the section 2.3, maintaining multiple software variants is difficult. Low level techniques, such as conditional compilation, and multiple files selected by means of directory paths, are heavily used in practice. Variation-RE activities are sometimes led in industry, but they are not identified as such. They correspond to tedious tasks done empirically, with poor tool support. The following table gives some examples of industrial problems and practices, as well as research topics.

3.6.1 Why?

- **Variation-RE-for-maintenance.** Implementing new variants often destructure the code. As mentioned in section 2.3.2, time pressure often prevent maintainers to do the right thing in the first place; patches are common practices. Variant restructuring is usually done afterwards, when a better knowledge of platform is available, or when patches lead to unreadable code.
- **Variation-RE-for-reuse.** Information about porting problems

across different platforms constitute a valuable knowledge. Unfortunately such knowledge is not represented explicitly, but disseminated in the mind of maintainers and in source code. Since portability issues concern virtually all software products, reusing porting experiences is important. Maintainers try to remember special tricks, talk with others, scan existing portable code, etc. Variation-RE can help to constitute a portability knowledge base by comparing platform libraries, and/or analyzing portable software products.

3.6.2 What?

Variation RE (What?)	examples
Reverse-engineering	extracting variation dimensions and configuration constraint from source code
Redocumentation	visualization of difference between variants
Restructuring	migrating from a file system representation, to a repository representation managed by a configuration manager like Adele; elimination of dead variants (those which are no longer used), reorganization of the code to gather all conditional compilation into “portable interface”, etc.
Modernization	translating components parametrized by means of macro-substitution into generic components (for instance C++ provides both variation-in-the-small implementation techniques).

3.6.3 How?

Some techniques used for evolution-RE can be reused, but specific techniques should also be designed for variation-visualization, variation-query-language, variation-pattern-recognition, variation-quality-metrics, etc.

4 The Champollion approach

Since the early 80’s, the objective of the Adele configuration management system was to provide better support for PITL, introducing new technologies into practice. More than ten years of industrial experience with this environment show the importance of the problems presented in section 2. While most CM tools are directly based on file system low level concepts, ADELE provides a powerful data model to describe and manage software artifacts. A special attention has been devoted to the communication between file systems and Adele repositories [EC94]. Adele is based on a coarse grain level and provided no specific support to integrate existing software products.

The Champollion project has been defined to cope with these limitations. Attention was first focused on the extraction of architectural data from source code. The goal were to provide a reverse-engineering-for-integration tool [EF89] and an architecture-query-language [Fav89]. Problems caused by the presence of preprocessor directives led us to consider variation-in-the-small problems. The necessity to take this new direction was also reinforced by the findings that there is almost no tool support to cope with these problems. Moreover, experiences with large software products convinced us that architecture could not be considered in isolation. We thus took a general approach.

This approach is based on the use of abstractions to model PITL artifacts. An abstract model, based on set theory has been proposed [Fav95a]. This model reflect the PITL classification presented in section 2 and address all related concepts. In [Fav95a], special attention was devoted to variation problems and technologies. CPP, the preprocessor of the C language, was taken as a case study. Existing PITS techniques proved to be useful to solve variation problems. We introduced variation-slicing, variation-specialization, and other variation-reverse-engineering techniques [Fav96]. All techniques have been

implemented in a prototype environment called APP. Given a file system structure containing CPP files, this environment produces graphical and textual views upon request. Current work is done to integrate these RE capabilities in the ADELE configuration manager and new research is oriented towards manufacture-reverse-engineering (currently attention is focused on of Make and its analysis).

5 Conclusion

Reverse-engineering-in-the-small is much more mature than reverse-engineering-in-the-large. Naturally this mirrors the difference existing between programming-in-the-small and programming-in-the-large. This does not mean that the latter is less important than the former! Since many maintenance problems are due to PRTL issues, more attention should be pay to reverse-engineering-in-the-large. Such activities are already led in industry but concepts are not formalized. Though fundamental, architecture-RE is not the only issue. We believe that adding RE capabilities to CM environment is a good strategy for the future, both for research and industry.

6 References

- [AT96] A. Auer, J. Taramaa ; "Experience Report on the Maturity of Configuration Management for Embedded Software", in Proc. of the 6th International Workshop on Software Configuration Management, Berlin, Germany, March 1996.
- [Arn93] R.S. Arnold; "Software Reengineering", IEEE Computer Society Press, ISBN 0-8186-3272-0, 1993, 675 pages.
- [CC90] E.J. Chikofsky, J.H. Cross; "Reverse Engineering and Design Recovery : A Taxonomy", in IEEE Software, January 1990, pp. 54-58
- [Dar92] S. Dart; "The Past, Present, and Future of Configuration Management", Technical Report CMU/SEI-92-TR-8, ESC-TR-92-8, Software Engineering Institute, Carnegie Mellon University, 31 pages
- [DCB93] S. Dart, A.M. Christie, A.W. Brown; "A Case Study in Software Maintenance", Technical Report CMU/SEI-93-TR-8, ESC-TR-93-185, Software Engineering Institute, Carnegie Mellon University, 51 pages.
- [EC94] J. Estublier, R. Casallas; "The Adele Software Configuration Manager", Chapter 4 of "Configuration Management", Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994
- [EF89] J. Estublier, J.M. Favre; "Structuring Large Versioned Software Products" in Proc. 13th International Computer Software and Applications Conference, Orlando, Florida, September, 1989, pp. 404-411,
- [Fav94] J.M. Favre; "Reengineering-In-The-Large vs Reengineering-In-The-Small", first SEI Workshop on Software Reengineering, Software Engineering Institute, Carnegie Mellon University, May 1994.
- [Fav95a] J.M. Favre; "Une approche pour la maintenance et la ré-ingénierie globale des logiciels", PhD dissertation, University of Grenoble, (France), 1995.
- [Fav95b] J.M. Favre; "The CPP Paradox", 9th european workshop on software maintance, Durham'95, Durham (UK) september 1995.
- [Fav96] J.M. Favre; "Preprocessors from an abstract point of view", submitted to the International Conference of Software Maintenance'96.
- [Gra92] J.E. Grass; "Cdiff: A Syntax directed Differencer for C++ Programs" in USENIX, Computing Systems, Vol. 5, N. 1, 1992.
- [GG96] B. Gulla, J. Gorman; "Experiences with the Use of a Configuration Language", in Proc. of the 6th International Workshop on Software Configuration Management, Berlin, Germany, March 1996
- [IBW93] P. Ingram, C. Burrows, I. Wesley; "Configuration Management Tools: a Detailed Evaluation", ISBN 0-903960-82-3, Ovum Ltd., London, 1993.
- [KS94] M. Krone, G. Snelting; "On the Inference of Configuration Structures from Source Code", Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994.
- [LS94] P. E. Livadas, D.T. Small; "Understanding Code Containing Preprocessor Construct", in IEEE Third Workshop on Program Comprehension, Washington, November 1994.
- [Mah94] A. Mahler; "Variants: Keeping Things Together and Telling Them Apart" , Chapter 3, in "Configuration Management", Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994.
- [Mal92] H.A. Muller and al.; "A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models" in ACM SIGSOFT, Software Engineering Notes, Vol. 17, No. 5, December 1992, Proc. of the 5th Symposium on Software Development Environments, pp. 88-98.
- [Sca94] W. Scacchi; "Technology Transfer", in Encyclopedia of Software Engineering, J.J. Marchiniak Editor, John Wiley and Sons, 1994, pp. 1323-1327.
- [SS93] R.W. Schwanke, V.A. Strack; "Configuration Management Problems and Architectural Integrity" , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993, pp. 225-228.
- [Sne95] G. Snelting; "Reengineering of Configurations Based on Mathematical Concept Analysis", Computer science report 94-02, Technical University of Braunschweig, Germany, January 1995, 28 pages.
- [SC92] H. Spencer, G. Collyer; "#ifdef Considered Harmful, or Portability Experience With C News" in USENIX, Summer 1992 Tecnical Conference, San Antonio (Texas), June, 1992, pp. 185-197.
- [TG95] E. Tryggeseth, B. Gulla; "Comprehensive Variability Modelling with the Proteus Configuration Language" , in Proc. of the 5th International Workshop on Software Configuration Management, Seattle, USA, April 1995.
- [VC92] K.P. Vo, Y.F. Chen; "Incl: A Tool to Analyse Include Files" in USENIX, Summer 1992 Tecnical Conference, San Antonio (Texas), June, 1992, pp. 199-208.

