

# Understanding-In-The-Large

Jean-Marie Favre  
IMAG Institute  
Laboratoire LSR/Adele  
BP53X, 38041 Grenoble Cedex 09, France  
jmfavre@imag.fr, <http://www-lsr.imag.fr/users/Jean-Marie.Favre>

## Abstract

*Developing and maintaining large industrial software products implies programming-in-the-large activities. Related concepts are usually represented in terms of low level features such as file system hierarchies, preprocessor files, makefiles, shell scripts, sccs archives, etc. Understanding the information embedded in such artifacts is an important but difficult task, especially with neither conceptual framework, nor tool assistance. To emphasize the importance of this issue, this paper makes the distinction between understanding-in-the-large and understanding-in-the-small. Using a conceptual classification, understanding-in-the-large problems are described in a structured way. Difficulties in building reverse-engineering-in-the-large tools are then analyzed and illustrated taking preprocessor files as a case study. The Champollion approach to these problems is briefly presented.*

## 1. Introduction

Understanding existing software representations is known to be a critical issue in the software engineering field. Currently, most research seeks to facilitate the understanding of “programs”, while software engineers have to understand “large software products”. Indeed, such products do not only consist of algorithms; they are complex structures of components existing in many different variants. Moreover, they evolve over time. In fact, understanding large software products requires programming-in-the-large concepts such as versioning, manufacturing, etc. to be taken into account.

The goal of this paper is to show that understanding programming-in-the-large information is an important but complex task and that reverse engineering tools can assist this process. While various authors address some specific aspects of this problem, a broad view is presented here.

It must be emphasized that understanding software

architectures is not the only issue. Since large software products are versioned, software engineers have to understand “program families”. This concept has been studied in the configuration management (CM) field. Unfortunately, in this domain, little attention is paid to the problem of understanding CM artifacts, although this is an important issue. On the one hand, modern CM environments gather much information but provide no useful way to understand it. On the other hand, the state of the practice is dominated by the use of low level tools (e.g., make, cpp, sccs); software representations associated with these tools are very difficult to understand.

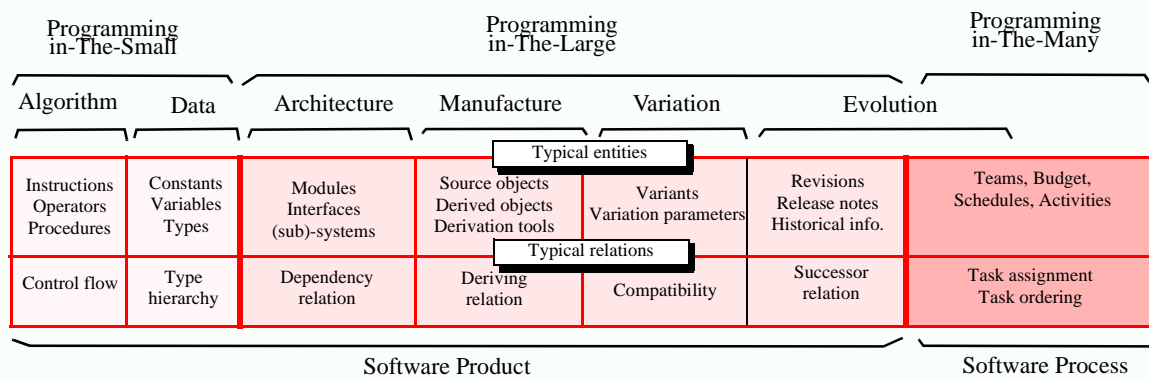
The Champollion project [5][6][7] aims to study the intersection between configuration management and reverse engineering. An important goal is to facilitate the understanding of programming-in-the-large information. This paper concentrates on this aspect.

Section 2 provides a classification of programming-in-the-large concepts and technologies. Section 3 uses this classification to present understanding-in-the-large issues in a structured way. Section 4 analyzes the difficulties in supporting understanding-in-the-large processes. Section 5 focuses on the understanding of CPP preprocessor files. Finally, section 6 briefly presents Champollion/APP, a reverse-engineering-in-the-large tool supporting the understanding of CPP files.

## 2. Programming-in-the-large

In a seminal paper entitled “Programming-in-the-large vs. Programming-in-the-small” [3], De Remer and Kron claimed that programming languages are well suited to describing algorithms and data structures, but not the structure of complex versioned software products. The main benefit of their work is that it shows that the software engineering discipline should not concentrate only on the traditional computer science concepts. Managing large software products involves a wide variety of problems, and research should not ignore them on the grounds that they are difficult to formalize and understand.

Figure 1 Programming-in-the-large concepts



## 2.1. Programming-in-the-large classification

In order to study the intersection between software understanding and programming-in-the-large in a precise and structured way, a precise classification of programming-in-the-large concepts is necessary. Note that we do not use the term “configuration management” because this term is mainly used as a rallying term. Indeed, it is not clear where the dividing line between what is CM and what is not CM lies. Furthermore, this frontier evolves over time because it depends on the functions provided by the tools.

We firmly believe that it is better to use a *conceptual* terminology rather than a terminology driven by random *technological* progress. So in [7], we introduced a precise and comprehensive classification based on the distinction between Programming-in-the-small, Programming-in-the-large and Programming-in-the-many (Figure 1). This classification is summarized below:

**Programming-in-the-small** (PitS) is concerned with two fundamental notions: **algorithms** and **data-structures**. Attention is devoted to the details of each individual software component.

**Programming-in-the-large** (PitL) focuses on concepts related to 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.

- **Manufacture** (or building). The *manufacture* description indicates how derived objects (e.g., binaries) are automatically produced by applying tools to source objects.

- **Evolution**: Software *evolution* cannot be avoided. This phenomenon is intrinsically connected to the notion of time. The point of interest in this paper is not the evolution process itself, but the impact of this process on the software product (e.g., revisions, historical information like change requests, release notes, etc.).

- **Variation**: The success of software often depends on its ability to operate simultaneously on various platforms and to accommodate a large variety of users and organizations. In such cases, different *variants* have to be implemented. Note that *variation* problems are not necessarily linked with the evolution concept. The existence of different variants can be independent from the time notion: the need for them can have been identified from the start of the 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 activities and resources (e.g. staff, time, budget, hardware, etc.). While PitS and PitL concepts are relative to the *software product*, Programming-in-the-many corresponds to the *software process*.

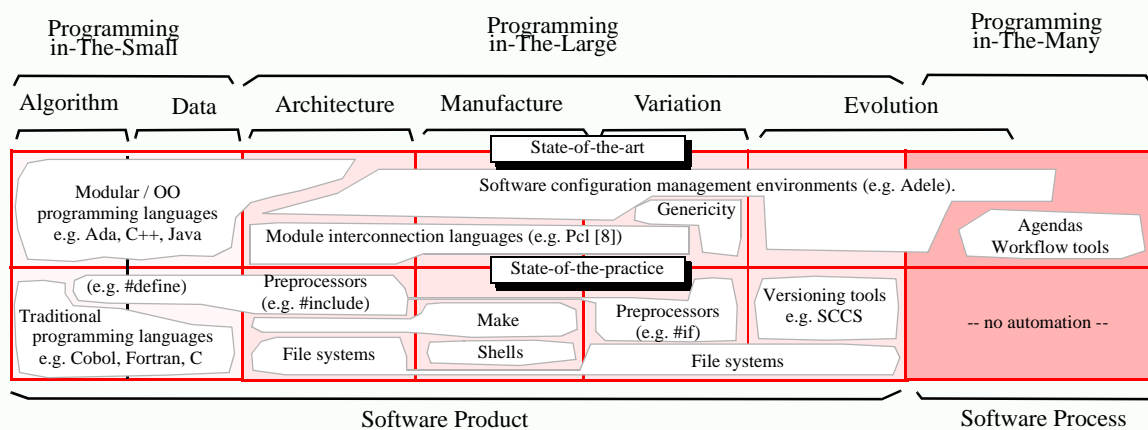
Obviously the classification presented above is an oversimplification. Though based on distinct concepts, all the fields are strongly connected [7]. For instance the intersection between architecture and variation (or evolution) gives rise to the configuration concept: a configuration is an assembly of multiple components (architecture) existing in different variants (variation) and/or revisions (evolution).

*Our objective is not to define new terms, but merely to lay emphasis on the topics usually masked by more prominent ones. Our goal is to highlight the manufacture, evolution and variation topics, which are usually overshadowed by programming-in-the-small and architecture.*

## 2.2. Programming-in-the-large technologies

Often confusion about the meaning of concepts is due to *technological* aspects. In fact, in almost all cases, the functions provided by a single tool cross the conceptual limits. For instance, while programming languages mainly support PitS concepts (i.e. algorithms and data structures), modular languages like Ada also represent PitL information (e.g. interfaces are architectural artifacts, generic

Figure 2 Programming-in-the-large technologies



components try to cope with variation problems). Similarly, most configuration management tools cover functions related to different fields, ranging from PitL to PitM. In spite of this complexity, the conceptual classification enables to draw a rough map of the programming-in-the-large technologies comparing the state-of-the-art and the state-of-the-practice (Figure 2).

### 2.3. State of the art

Nowadays, a wide variety of tools address various PitL functions. The overall trend is to integrate concepts and/or technologies developed in other fields into CM systems: object orientation; deductive databases; feature logics, etc. Despite these innovations, no specific attempt has been made assist explicitly the understanding of modern CM artifacts. Furthermore, the usage of state-of-the-art CM tools remains rare in the software industry.

### 2.4. State of the practice

Let us concentrate on the state-of-the-practice. The corresponding technologies are less exciting.

- **Architecture.** Most existing software products lack explicit *architecture* description. Traditional programming languages (e.g. Cobol, C) provide no PitL support. The notion of interface is usually encoded in terms of textual inclusion (e.g. #include "stdio.h"). In most cases the only concrete representation of systems and subsystems is the hierarchical organization of source codes in terms of directories and sub-directories (i.e. ada/link/parser). There is no explicit representation of inter-system connections.
- **Manufacture.** Software *manufacture* is automated by means of shell scripts, job control languages or makefiles. Make uses architectural information (the dependency relation) and derivation rules to automate the manufacturing process. Despite many drawbacks, Make is probably one of

the most popular software engineering tools.

- **Variation.** Software *variants* can be represented at two levels of granularity. On the one hand, different file name conventions can be used to distinguish existing variants (e.g. parser-msdos.h, parser-pdp11.h) or different directories can be used (e.g. it/msdos/parser.h and it/pdp11/parser.h). In the latter case, the selection of source objects is realized through a tool's path options during the manufacturing process (e.g. -I or -L for Unix compilers). On the other hand there are many similarities between variants, all variants are gathered into a single file, specific parts being included between conditional compilation directives (e.g. #ifdef pdp11 ... #endif)
- **Evolution.** Similar techniques 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 (e.g. parser-1.h, parser-2.h,...). Tools like SCCS are used to save disk space, and gather historical information about software changes (asking for a "comment" at check-in time).

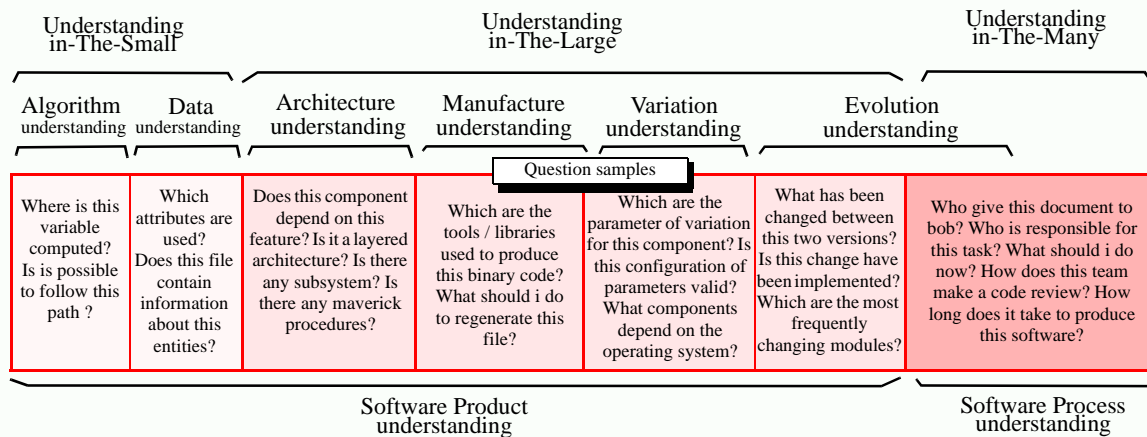
*Note that most of the state-of-the-practice tools in use today were designed before or during the 70s!!!* For instance preprocessor technology has been popular since the 60s; CPP, the C preprocessor, was defined in the latter part of the 70s; Make has been in use since 1975; SCCS since 1974, etc. Moreover, some estimations suggest that *less than 10% of programmers on the planet use a modern configuration manager* [12]. Old PitL technology is still largely predominant.

### 2.5. Synthesis

There is no direct mapping between concepts and software artifacts: an artifact may contain different kinds of conceptual information; conversely a single concept can be represented by different kind of artifacts.

Figure 3

## Understanding-in-the-large



### 3. Understanding-in-the-large

The term “software understanding” is very general. It can be refined using the conceptual and technological classifications presented in the previous section.

#### 3.1. Conceptual classification

**Understanding-in-the-small** (UitS) refers to the understanding processes focusing on programming-in-the-small concepts, that is to say on algorithms and data structures. **Understanding-in-the-large** (UitL) refers to the understanding process which concentrates on programming-in-the-large concepts, that is to say on architecture, manufacture, variation, or evolution.

This conceptual classification, illustrated in Figure 3, is useful essentially when a “top-down” approach to software understanding is used. In this case, the software maintainer is looking for a conceptual information of a specific nature and he tries to identify which parts of the software are related to his problem.

Note that during this process, he may scan different kind of artifacts. For instance, if the goal is to discover architectural information, browsing makefiles, source code and/or file systems is possible.

#### 3.2. Technological classification

Instead of using the *conceptual* classification, we can consider the *technological* dimension and define **program understanding, file system understanding, makefile understanding, preprocessor file understanding**, etc.

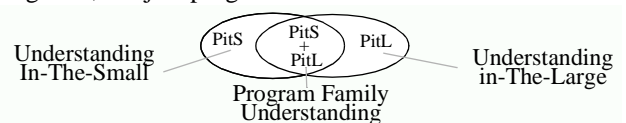
This technological classification is useful essentially when the maintainer uses a “bottom-up” approach. In this case, he tries to synthesize conceptual information from the examination of a specific artifact. During this process, he

can discover different kind of concepts. For instance if he reads a makefile, he will learn about the software architecture and software manufacture (see Figure 2 on the previous page).

#### 3.3. Program family understanding

Clearly, these classifications do not define strict limits. They merely identify different kind of concepts and artifacts. As pointed out in section 2.1, many intersections are meaningful. For instance, some software engineering tasks require understanding of the *evolution* of the *architecture* (e.g. did we add new dependencies?), or the *variation* of the *manufacturing process* (e.g. which tools are used on each platform?).

Similarly, the intersection between understanding-in-the-large and understanding-in-the-small is not empty. Understanding PitS information can require understanding PitL information and vice versa. In this case, we will use the term “**program family understanding**”. This term derived from the CM terminology, shows explicitly that software engineers sometimes have to understand versioned programs, not just programs.



#### 3.4. Importance

*Understanding-in-the-small.* Obviously, every programmer must understand PitS artifacts, at least the part relevant to his work. Without this understanding, the software could not be maintained.

*Program family understanding.* Each programmer should also have a partial understanding of the *architecture*

of the system, at least to control the interaction with the other components. He should be able to understand the *evolution* of the artifacts, at least when he has to write a release note synthesizing all the changes actually implemented since the last major release. When an algorithm or a data structure is platform-dependent or user-dependent, the programmer has to maintain and understand multiple *variants*. This implies at least a partial understanding of configuration constraints, variation parameters, etc. All these tasks involve PitS and PitL concepts simultaneously.

*Understanding-in-the-large.* Many other software engineers concentrate on PitL information without a detailed knowledge of the software. These are, for instance, “software architects” or “configuration managers”. These roles have been introduced in large organizations to support PitL activities. Since they do not implement the software themselves, they are confronted with understanding-in-the-large problems. For instance, a software architect has to check if the actual architecture complies with his conceptualisation. This is especially true for the maintenance of legacy systems. One important challenge in the CM community is facilitating the transfer of CM technology [14]. During the process of reengineering PitL information (see [8] for an example of such a process), understanding-in-the-large is of paramount importance. Without a full understanding of variation aspects, describing a complex multi-variant architecture with a new tool is not possible.

### 3.5. Difficulties

Understanding PitL information is a difficult task, especially when old PitL tools are used. There are many reasons for this:

- **Lack of abstraction.** In the PitL field, people do not use abstractions to describe what they do. They speak and think in terms of “Make targets”, “directory paths”, etc. Communicating and reasoning is not easy in such a context.
- **Lack of PitL documentation.** Often, existing PitL artifacts are not documented or the documentation is not up-to-date.
- **Encoded PitL information.** Most PitL tools were designed without readability in mind. Expressions are character-based rather than token-based. For instance, deciphering complex shell expressions is really painful.
- **Scaling problem.** Many researchers in computer science believe that understanding *a* makefile and *few* variants is not an actual problem. This is true. The problem is that *large* software products like X/Window system are composed of hundreds of makefiles and variation parameters. For instance, the cku-5a implementation of ‘kermit’ is running on more than 298 different software platforms! The

algorithmic complexity of this small communication software product is low when compared to its PitL complexity. Understanding-in-the-large is the main problem in this case.

- **Information dissemination.** Often PitL information is scattered among many artifacts. For instance, understanding a single line of a preprocessor file may require browsing through many files disseminated over all the file system hierarchy. Sometimes, the information needed is not even available at the maintenance site. For instance, understanding a small piece of a portable code can require to browse many include files on many platforms. This is usually not possible for practical reasons.

- **Language mixture.** Understanding a piece of PitL information involves dealing simultaneously with different languages. For instance, the behavior of a file inclusion directive (e.g., #include “libg.h”) is typically controlled by means of compilation options (e.g. -I mylib) which are themselves under the control of makefiles, whose parameters are set by shell scripts, etc. Software engineers have to deal with all these languages during an understanding-in-the-large process. The worst of it is that these different languages do not even use the same conventions to represent the same concepts. For instance, the meaning of regular expressions is not always the same!

- **Unclear tool semantics.** Indeed, in many cases, PitL tool semantics is not even clear. “Unexpected behavior” is common in the PitL domain. For instance, even experienced C programmers have difficulties in understanding how the CPP preprocessor works in specific situations.

- **Irrationality of variation and evolution.** Understanding an artifact often requires previous knowledge about the application domain. For instance, a programmer can understand a payroll program because he comprehends this logical concept and he has a previous knowledge about this domain. He can understand the *architecture* of such a system if he knows that this concept can be decomposed into various functions. By contrast, often the *evolution* and *variation* of a software system are not governed by any rationale. These aspects are therefore difficult to “understand”. For instance, the existence of two software variants could be: “in New York a developer called this library function ‘strcp’ while the same function was called ‘strcpy’ by another programmer in Berkley”. Such anecdotal facts are not necessarily connected with technical aspects: “the functionality X is needed by organization O<sub>1</sub> but organization O<sub>2</sub> in Tokyo prefers Y”. Nor software evolution is not either rationally driven: “finally O<sub>2</sub> prefer functionality Z; this is why there are two releases”. Indeed, maintainers have to cope with arbitrary sets of requirements. Faced with variation or evolution artifacts, they cannot find explanations, they are reduced to taking notes.

### 3.6. Consequences

All the difficulties described above make understanding-in-the-large processes a demanding task. Indeed, PitL knowledge about software is difficult to share, to transmit and to reconstitute after being lost. As a result, only a few people are able to explain some PitL aspects of the software.

This is typically the case for porting problems. In many organization, there are only one or two “gurus” who know precisely (1) what all the differences between operating system libraries, (2) how to cope with these problems, (3) what is the corresponding impact on source codes or manufacturing processes.

In fact, when numerous variants are maintained, almost nobody truly understands all the software. The maintainers are forced to use “as-needed” comprehension strategies and they make assumptions about the software. Maintenance becomes a “hit or miss” process [16].

Without a comprehensive understanding, patches are applied on patches, thus increasing the complexity of PitL artifacts. Program families increase in size. Code is steadily added, but never removed. When a platform or a specific functionality becomes obsolete, removing the specific code for it is difficult. Interweaved variant structures are most awkward in performing such operations. For instance, is there somebody on the planet able to look at gcc, X-window or kermit software systems and ensure that every piece of code is used on at least one running platform?

### 4. Support for understanding-in-the-large

Assisting software understanding processes is an important goal, but today, most research work concentrate on understanding-in-the-small support. Thanks to this effort, different techniques have been devised: algorithm slicing, data structure visualization, algorithm cliché recognition, etc. Indeed, the reason that make PitL artifacts hard to understand, are the same reasons that slow down the development of reverse-engineering-in-the-large tools:

- **Lack of abstraction.** Any reverse engineering tool is based on the use of abstractions. But, as pointed out before, there is a lack of abstraction in the PitL domain. Even though few modern CM systems are based on some abstractions (feature logics, and/or graphs, etc.), their use is usually oriented towards a specific (set of) system(s). Moreover, they are not powerful enough to represent the full complexity of existing PitL artifacts.

- **Lack of available techniques.** The development of the reverse-engineering-in-the-small field built on the existing PitS technology, especially on the well-founded compilation technology. Which techniques could we reuse to speed up the development of RitL techniques?

- **Concept mixture.** Since all PitL aspects are connected,

addressing manufacturing problems implies considering variation problems, and so on. Hence, to build a useful RitL tool, many interweaved concepts should be taken into account simultaneously. This makes the development of such a tool even more complex.

- **Language mixture.** These connections also arise from a technical point of view. Since preprocessor files depend on makefiles which can depend on shell scripts (and so on), a single reverse engineering tool is not sufficient. All artifacts should be analyzed to solve the whole problem.

- **Unclear tool semantics.** To build a confident tool based on safe transformations or inferences, the semantics of the artifact should be clear. In the PitS domain, describing the semantics of a programming language is known to be important. What about the formal semantics of file systems, shells scripts, makefiles, sccs archives or even cpp files?

As a consequence UitL tool support is quite limited, especially when compared with the UitS domain. To understand PitL artifacts maintainers rely on elementary tools like *emacs*, *grep* and *diff*. These tools offer a basic but indispensable assistance. Unfortunately, since they do not take into account the semantics of PitL artifacts, they can give misleading results.

*Clearly, designing confident reverse-engineering-in-the-large tools is necessary.*

## 5. Understanding CPP files

To illustrate the concepts and problems presented above, let us take as a case study the problem of understanding preprocessor files, in particular CPP files.

### 5.1. CPP - the C preprocessor

CPP is the preprocessor of the C and C++ languages. This very low-level tool is very popular among programmers. This popularity is probably due to (1) its *simplicity* and (2) its *flexibility*.

The *simplicity* of CPP is due to the fact that it provides only 4 constructs, namely file inclusion (`#include`), macro definition (`#define`), macro substitution and conditional compilation (`#if`). Furthermore, CPP does not deal with complexe entities, but only with strings. From a technical point of view CPP is thus a very simple tool.

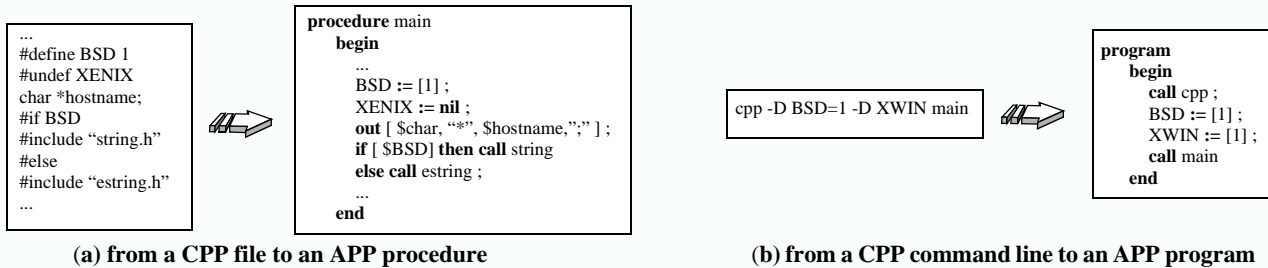
The *flexibility* of CPP is a consequence of its simplicity. Since this is a string-based tool, it can be used with any kind of *file*, including programs, nroff documents, makefiles, shell-scripts, etc. It is thus important to distinguish two understanding problems: CPP file-understanding and CPP program-understanding.





Figure 6

## From CPP to APP



(a) from a CPP file to an APP procedure

(b) from a CPP command line to an APP program

definitions, which can depend on file inclusions, depending in turn on path options, etc.).

- (3) **Lack of available techniques.** CPP seems to be a very specific tool. This isolation makes it difficult to reuse techniques from other domains.
- (4) **Unclear tool semantics.** Without a precise definition of CPP semantics, building reliable tools is not possible.

*In fact, building confident CPP reverse engineering tools is a challenge.*

## 6.2. The Champollion/APP approach

While the goal of the Champollion project is to study understanding-in-the-large and reverse-engineering-in-the-large issues, special attention has been devoted to CPP, giving rise to the Champollion/APP approach. As shown below, this approach give answers to problems (1) to (4).

**APP, an Abstract PreProcessor** has been defined to cope with problem (1). APP is semantically equivalent to CPP. It has been designed for reverse engineering purposes only. It is used as a means for describing internal representations for CPP files. This abstract language has no concrete syntax: it just describes abstract syntax trees equivalent to CPP files.

The APP language takes into account all CPP features to cope with problem (2).

In fact, though semantically equivalent to CPP, APP is based on traditional PitL concepts and terminology (variables, instructions, procedures, etc.). Using these abstractions turns out to be a very powerful method to cope with (3): many existing program analysis techniques can be used to solve CPP file understanding problems!

Finally, to deal with (4) we rigorously defined the denotational semantics of APP (and thus of CPP). The formal definition takes a few pages. It was the basis of confident static analysis techniques (data flow analysis, partial evaluation, slicing, etc.)

## 6.3. From CPP to APP

The first step of the reverse engineering process is the generation of APP abstract programs from CPP concrete

files. This abstraction schema, defined rigorously in [7], is illustrated in Figure 6 (a few keywords are used for the sake of clarity).

The basis of the abstraction schema is that CPP directives correspond to APP statements (see Figure 6.a). Macro definitions translate into variable assignments. Each piece of text included in a CPP file translates into an output statement. Conditional compilation directives naturally convert to conditional statements. Since a CPP file is a sequence of CPP directives and pieces of text, the correct abstraction for a file is a procedure (i.e. a sequence of statements). As a consequence, a file inclusion is a procedure call!

CPP invocation lines are also converted for uniform treatment (see Figure 6.b). The procedure call to the main file is made explicit as well as macro definitions. A procedure named `cpp` is called to reflect the fact that the execution behavior depends on the `cpp` executable file which can be different on each platform. This procedure contains an assignment for each predefined macro (such as `sun`, `hp`, etc.)

## 6.4. APP program analysis

The main benefit of using APP instead of CPP is that APP refers to PitS concepts. This makes it possible to adapt existing program analysis techniques to the CPP specificities. Since CPP files are equivalent to (APP) programs, program analysis techniques can be used to support reverse engineering and understanding of CPP files. The value of control-flow analysis, data-flow analysis, slicing and partial evaluation has been shown in [6].

For instance, data-flow analysis makes it possible to automatically generate procedure signatures (via the computation of reaching definitions and live variables). Such signatures are very interesting from a maintainer point of view: they indicate which macros should be set to configure a specific set of files, which macros are changed by the inclusion of a file, etc. Similarly, partial evaluation techniques automate partial readings [6].

## 7. Related work

Most reverse engineering tools concentrate on reverse-engineering-in-the-small and architecture-reverse-engineering. Only a few contributions take into account manufacture, variation and/or evolution concepts. Let us concentrate on them.

*PCL reverse* is a component of *Proteus*. The latter is a forward engineering CM environment generating makefiles from a configuration model described with the PCL language [8]. The PCL browser facilitates the comprehension of PCL models. The PCL-reverse tool is a limited reverse-engineering-in-the-large prototype allowing automatic construction of rudimentary PCL models for existing software.

Similarly, *Cc2Cov* is a limited reverse engineering prototype automating the integration of CPP source files into CoV, a research CM environment [14]. *HiCoV* [15] is a visual formalism tailored to the visualization of CoV configuration constraints. It aims to facilitate the understanding of large sets of constraints.

*Recs* is [13] a variation-reverse-engineering tool based on features logics. More specifically, it concentrates on CPP file understanding as defined in section 5.2. In other words, *Recs* has no knowledge of the file contents. Actually, it takes into account only conditional compilation directives, ignoring macro definitions, macro substitutions, and file inclusion directives. It can support understanding-in-the-large via the visualization of a lattice representing in a compact form the relation “variation parameter” - “text fragment”.

*Seesoft* [4] is a software visualization tool which allows statistics to be displayed for each line of code in a compact form. In particular, it can display evolution information, such as the age of the code, or the difference between two versions. *SeeSys* [1] provide roughly the same functionality, but it is adapted to the visualization of hierarchical structures (e.g., the software structure in terms of directories).

*Gaze* [11] is a visualization tool displaying the evolution of software architectures at a coarse granularity level.

## 8. Conclusion

Programming-in-the-large issues are important. Even if this field is not as mature as the programming-in-the-small field, it should not be ignored. In this paper we have introduced a classification taking into account a large number of software engineering concepts. Our goal was not to introduce new terms, but simply to emphasize certain aspects such as variation problems. Indeed, understanding-in-the-large is an old problem, but an emerging research theme.

## 9. References

- [1] M.J. Baker, S.G. Eick; “Visualizing Software Systems”, in Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 59-67
- [2] T.J. Biggerstaff, B.G. Mitbender, D.E. Webster; “Program Understanding and the Concept Assignment Problem”, in Communications of the ACM, May, 1994, pp. 72-83.
- [3] F. DeRemer, H. Kron; “Programming-in-the-Large vs. Programming-in-the-Small” in IEEE Transactions on Software Engineering, Vol. 2, N. 2, February 1976, pp. 80-86.
- [4] S.G. Eick, J.L. Steffen, E.E. Sumner; “Seesoft - A Tool For Visualizing Line Oriented Software Statistics”, in IEEE Transactions on Software Engineering, Vol. 18, N. 11, November 1992, pp. 957-968
- [5] J.M. Favre; “A Rigorous Approach to the Maintenance of Large Software” in Euromicro Working Conference on Software Maintenance and Reengineering, (Berlin), IEEE Computer Press, March 1997.
- [6] J.M. Favre; “Preprocessors from an abstract point of view” in Proc. of the International Conference on Software Maintenance-1996, also in Proc. of the International Working Conference on Reverse Engineering, November 1996.
- [7] J.M. Favre; “Une approche pour la maintenance et la ré-ingénierie globale des logiciels”, PhD Thesis, Université Joseph-Fourier Grenoble-I, Grenoble (France), octobre 1995.
- [8] J. Floch, B. Gulla; “Enabling Reuse with a Configuration Language”, in Proc. of the 4th International Conference on Software Reuse, Orlando (Florida), April 1996
- [9] J.E. Grass; “Cdiff: A Syntax directed Differencer for C++ Programs” in USENIX, Computing Systems, Vol. 5, N. 1, 1992.
- [10] J.M. Gravley, A. Lakhota; “Identifying Enumeration Types Modeled with Symbolic Constants”, in International Working Conference on Reverse Engineering Nov. 1996, pp. 227-236
- [11] R.C. Holt, J.Y. Pak ; “GASE: Visualizing Software Evolution-in-the-Large”, in Proc. of the 3rd Working Conference on Reverse Engineering, Nov. 1996, pp. 163-168
- [12] P. Ingram, C. Burrows, I. Wesley; “Configuration Management Tools: a Detailed Evaluation”, ISBN 0-903960-82-3, Ovum Ltd., London, 1993.
- [13] M. Krone, G. Snelting; “On the Inference of Configuration Structures from Source Code”, Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994.
- [14] B.P. Munch; “Versioning in Software Engineering Database : the Change Oriented Way” , PhD dissertation, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, 1993.
- [15] B.P. Munch ; “HiCoV: Managing the Version Space”, in Proc. of the 6th International Workshop on Software Configuration Management, Berlin, Germany, March 1996. COV, EPOS, EPOSDB
- [16] H. Spencer, G. Collyer; “#ifdef Considered Harmful, or Portability Experience With C News” in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 185-197.
- [17] “unifdef”; unix man pages.
- [18] K.P. Vo, Y.F. Chen; “Incl: A Tool to Analyse Include Files” in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 199-208