Preprocessors from an Abstract Point of View

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

It is well known that the extensive use of preprocessors can lead to serious maintenance problems. However, these tools are still heavily used by developers and maintainers to implement software variants. Without adequate tools, understanding complex preprocessor files is a really tedious task. Problems are difficult to formulate and seem very specific. This paper shows that considering preprocessors from an abstract point of view can lead to surprising results. The use of abstractions considerably increases problem comprehension and solutions can be derived directly from existing techniques like slicing, program specialization or interprocedural data flow analysis. The preprocessor of the C language (CPP), is taken as a case study, and the functionalities of Champollion/APP, a tool for preprocessor maintenance, are briefly described.

1. Introduction

In a seminal paper entitled "*Programming-in-the-large vs Programming-in-the-small*", De Remer and Kron argued that programming languages are well suited to describe algorithms but not the structure of complex versioned software products. Since then, the separation of concerns between Programming-in-the-large (PITL) and programming-in-the-small (PITS) has been considerably reinforced and the corresponding research lines have evolved in parallel.

As shown in this paper, preprocessors are pragmatic tool which bridge the gap between PITS and PITL. On the one hand, most preprocessors have been designed by PITS practitioners and are themselves (degenerated) languages. On the other hand, preprocessors try to partially solve a major PITL problem: software variation. Unfortunately, their extensive use lead to severe maintenance problems.

Section 2 briefly presents variation problems. Section 3 is devoted to the presentation of the C preprocessor, from a concrete point of view. A small scenario is used to show how preprocessor use can lead to incomprehensible structures, and pragmatic problems are described. In section 4, CPP is studied from an abstract point of view. Using this abstraction, problems are reformulated in section 5, and solutions are shortly described. Finally, in section 6, the Champollion/APP prototype is presented.

2. Variation problems

Variation problems are not specific to software. The main concepts are first presented in abstract terms. Then, the special case of software is studied, putting emphasis on practical issues.

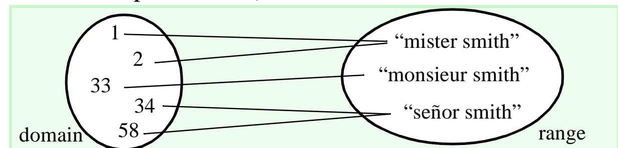
2.1. Variation from an abstract point of view

In order to formalize variation problems, an abstract model based on set theory has been defined in [1]. The presentation of this model is outside the scope of this paper. Only a few basic notions are informally presented here.

Let suppose that an *object* (in the broad sense of the term) is used in a precise *context*; for instance the sentence "mister smith" is used in the USA (this is the context).

The existence of different contexts usually gives rise to a problem: if a single entity is not adapted to the context *variation*, different *variants* are needed. For instance "monsieur smith", "señor smith" and "mister smith" are 3 variants. They are necessary if we want to deal with the country variation.

The collection of variants is called a *generic object*. To put it more precisely, from an abstract point of view, a generic object is a *function* (in the mathematical sense of the term), which maps formal context descriptions into variants. For instance, the following function, referred to as g_name, is a generic object mapping a country description (coded by means of its phone code) into a variant.



Proc. of the International Conference on Software Maintenance, IEEE Computer Press, November 1996, pp. 329-338
also in Proc. of the International Working Conference on Reverse Engineering, IEEE Computer Press, November 1996. pp. 287-296

The *domain* of this function is {1,2,33,34,58} while the *range* is the set of possible variants.

As state above, functions are mathematical abstractions corresponding to generic objects. In practice, the critical issue is to choose a good representation for generic object. Below, two possible representations are presented for the g_name generic object (using a functional language). They are semantically equivalent, i.e. they correspond to the same abstraction.

```
function g_name (x : integer) =
case x of
   1 : "mister smith"
   2 : "mister smith"
   3 : "monsieur smith"
   34 : "señor smith"
   58 : "señor smith"
end
```

```
function g_name (x : integer) =
 (case x of
    1,2 : "mister"
    33 : "monsieur"
    34,58 : "señor"
 end)
 + " " + "smith"
```

In fact, since generic objects have to evolve, maintainability is one of the main qualities required for their representation. Let us suppose for example that, the "s" of smith has to be capitalized. Only one modification is needed to implement this change on the second representation. In contrast, if the first representation is used, this operation has to be successively applied to variants. This problem, due to the duplication of information between variants, has been named "*the multiple maintenance problem*" [2]. Sharing information avoids this problem, but on the other hand, increases the representation complexity and decrease understandability (as an illustration, a unreadable example is presented in section 3.3).

Thus, the key issue when representing generic objects, is to ensure maintainability by means of a compromise between understandability and duplication effort.

2.2. Software variation in practice

Software products are very complex objects. They are used to make the interface between user needs and computer capabilities. More precisely, software products are used in a two-aspect context: the *technical context* which corresponds roughly to the software and hardware platform, and the *logical context,* which correspond basically to organization and user needs.

Clearly, the logical and the physical contexts are subject to variations. Software products must be adapted, that is to say, different variants are needed. The problem is thus to represent generic software products. Since they have to evolve, maintainability is a key concern.

Technical variations. In practice, technical variations give rise to porting activities. Constraints associated with such activities are very strict [3]. Delivery times are usually very short. Software products must often be adapted to new or foreign platforms, unknown by the maintainers. Clearly, standards like POSIX help to limit variations between systems, but, as pointed out by Tilbrook and Crook, adherence to standards only partially resolves porting problems [4].

Logical variations. While porting is an important aspect, other goals lead to variations. Software products should be adapted to meet organization needs, users preferences, etc. [5-7].

The multiple maintenance problem is especially acute for software products. Obviously, it is impossible to represent variants by making n copies of the software. Code sharing is compulsory.

Different technologies have been designed to cope with software variations at different levels of granularity. A survey is in given in [1], ranging from special tricks with files systems to genericity in modern programming languages or software configuration tools. Here we focus our attention on one of the worst but nevertheless effective technologies: preprocessors.

3. CPP from a concrete point of view

CPP, the *C P*re*P*rocessor is taken as a case study. Since it provides basic features like macro substitution, textual inclusion and conditional compilation, it is relatively representative of other textual processors, and a large proportion of what is said in this paper should be applicable to other text processors.

3.1. Importance of CPP

Why did we choose CPP? CPP is widely used both in industry and research. C and CPP are fundamentals in the UNIX operating system. CPP is also used with other programming languages like Pascal dialects and C++. Its wide availability means that it is also employed with text files, LaTeX documents, makefiles, X resources files, etc.

Finally, although CPP is a very low-level tool, its significance has been mentioned in workshops on software configuration management [2,3,8-10] and on software reuse [11]. Problems due to its use have also been reported for large software products [4,12] but only few tools have been developed specifically for it [13,14]. Recently, various independent research programs have shown a growing interest in CPP [1,5-7,15-19].

3.2. CPP features

CPP is a textual preprocessor. It means that, before the compilation phase, a textual transformation is applied to the program sources (or any text file). CPP provides only a few low level features. For further information, the reader is invited to consult a C programming language manual or [20].

The **file inclusion directive** (#INCLUDE "<FILENAME>") allows the composition of multiple files. It takes a filename as an argument and temporarily changes the input in order to process the specified file.

The **macro substitution** mechanism provide roughly similar services but it deals with string variables assigned by means of **macro definition** directives (#define <macroname> <string>). The #undef <macroname> directive can be seen as a special case. It associates a special macro value, let's say "undefined", to the specified macro name. On invoking CPP, some macro name assignments can be specified. The undefined value is virtually assigned to other macro names.

Conditional compilation directives take the form of a #IF *<condition>* *<cpptext1>* #ELSE *<cpptext2>* #ENDIF sequence. After macro substitutions, *<expression>* must be a integer constant expression. 0 stands for false as in the C language.

Indeed file inclusion, conditional compilation and macro substitution are closely related and are often largely interleaved: conditional expressions depend on macro values; macro values depend on conditional compilation and file inclusion; at least if these constructions contain macro definitions.

3.3. A simple scenario

Preprocessor are usually used with large and complex software products. Explaining the behavior of these artifacts with respect to variation problems is very difficult: a lot of technical aspects arise at the same time and make the overall behavior particularly unclear. Furthermore, preprocessor directives are usually spread among thousands of code lines distributed in numerous files and directories, making it unpractical to give illustration of complex systems.

To isolate and illustrate variation problems, we have developed in [1], a scenario based on a very small system. We claim that this scenario is representative of the behavior of large software product with respect to variation and that it makes explicit most of related problems. The seven CPP files presented on the next page constitute the whole system! They represent a generic letter usable for conference mailing.

Indeed, like other software products, this scenario itself had to evolve due to changing requirements and unplanned context evolutions! Originally, it was designed for a French context (a French Ph.D. dissertation [1]). Since it proved to be useful and valuable, it was adapted to a foreign and unexpected platform (this was for a slide presentation in an English speaking workshop [21]). As for porting activities, the constraints associated with this adaptation were very stringent:

• a very short dead line (few hours before taking the plane to the UK),
• a poorly qualified maintainer with respect to the new platform (the author of this paper has a limited knowledge of the English language),
• platform with very different characteristics and with specificities (irregularities in French and English language are numerous and obviously different).

This led to further destructuring of the system, and decreased system reliability, understandability, etc. At this point, it is important to emphasize that originally, the system was very carefully designed, with a strong emphasis on evolution. However, this example proves that it is impossible to anticipate all evolution (here patches are mainly due to linguistic irregularities in different languages).

Now, let suppose that you play the role of the new maintainer. The system is supplied as it stands, and before implementing a new change, the maintainer must understand the system structure. As in bad cases (that is to say, in most cases), historical information about the early days of the system are not available or are not up to date (in fact they are fully described in [1], but in French). Probably, some portions of code are really obscure, and the overall structure is unclear. The maintenance challenge is precisely to deal with such problems.

3.4. CPP problems

There are a lot of problems associated with the use of CPP [21] and different authors have described some of them, usually from a specific or pragmatic point of view [12,15,20,21]. An extensive list is given in [1], as well as a classification.

The goal of this section is to informally describe some of these problems. In section 5, each of them will be reformulated from an abstract point of view, providing solutions at the same time.

The behavior of CPP is sometimes confusing or can lead to errors very difficult to discover. Problems ***P1*** and ***P2*** are representative.

P1. Any macro occurrence appearing in a conditional expression will be substituted by 0 if the macro is not defined. This default value is arbitrary and may cause surprising results when used in arithmetic expressions (for example A<B+3 is true if A and B are not defined!?!).

P2. Macro occurrences do not expand at macro definition time. The reader usually assumes the contrary. For instance, let us consider the following sequence.

```
#define A 1
#define B A+2     A is not expanded here, macro B stand for "A+2"
#define A 10
B
```

When reading the B definition, the reader naturally concludes that B stands for 3. He makes a mistake! The B

File g_possessive

```
#if GROUPNB == 1
#if SUBJECTNB == 1
#if ENGLISH
"my"
#else
"m"
#endif /*ENGLISH*/
#elif SUBJECTNB == 2
#if ENGLISH
"t"
#else
"your"
#endif /*ENGLISH*/
#elif SUBJECTNB == 3
#if ENGLISH
"his"
#else
"s"
#endif /*ENGLISH*/
#endif /*SUBJECTNB*/
#if ENGLISH==0
#if OBJGENDER==1 && OBJNB==1
"on"
#elif OBJGENDER==0 && OBJNB==1
"a"
#elif OBJNB>1
"es"
#endif /*OBJGENDER...*/
#endif /*ENGLISH*/
#elif GROUPNB > 1
#if SUBJECTNB==1||SUBJECTNB==2
#if SUBJECTNB == 1
#if ENGLISH
"our"
#else
"n"
#endif /*ENGLISH*/
#elif SUBJECTNB == 2
#if ENGLISH
"your"
#else
"v"
#endif /*ENGLISH*/
#endif /*SUBJECTNB...*/
#if ENGLISH == 0
#if OBJNB==1
"otre"
#elif OBJNB>1
"os"
#endif /*OBJNB==1*/
#endif /*ENGLISH*/
#elif SUBJECTNB == 3
#if ENGLISH
"their"
#else
"leur"
#define PLURAL (OBJNB>=2)
#include "g_termination"
#endif /*ENGLISH*/
#endif /*SUBJECTNB=...*/
#endif /*GROUPNB...*/
```

File g_title

```
#ifndef NUMBER
#define NUMBER 1
#endif
#if ENGLISH==0
#    define GROUPNB 1
#    define SUBJECTNB 1
#    define OBJGENDER GENDER
#    if NUMBER>1
#        define OBJNB 2
#    else
#        define OBJNB 1
#    endif
#    include "g_possessive"
#    include "g_feudal_title"
#    define PLURAL (NUMBER>1)
#    include "g_termination"
#else
#    if GENDER==1
#        if NUMBER>1
             "messrs"
#        else
             "mr"
#        endif
#    elif GENDER==0
#        if MARRIED==1
#            if NUMBER>1
                 "misses"
#            else
                 "miss"
#            endif
#        else
             "mrs"
#        endif /*MARRIED*/
#    endif
#endif
```

File g_letter

```
#include "g_config_people"
#include "g_config_conference"
NAME, ADDRESS
#if ENGLISH
"Dear"
#endif
#include "g_title"
NAME
#if ENGLISH
"We are pleased to be able to tell you"
"that the referees have accepted your"
"paper entitled " TITLE "submitted to"
#else
"Nous avons le plaisir de"
#if NUMBER>1
"vous"
#else
"t'"
#endif /*NUMBER>1...*/
"annoncer que les rapporteurs ont"
"accepte le papier intitule " TITLE
"soumis a"
#endif /*ENGLISH*/
CONFNAME
#if ENGLISH
"We are looking forward to seeing you in"
#else
"Nous vous attendons a "
#endif
CONFCITY
```

File g_config_conference

```
#if DURHAM95
#define CONFNAME "the Durham'95 workshop"
#define CONFCITY "Durham"
#define ENGLISH 1
#endif
#if ICSM96
#define CONFNAME "the ICSM'96 conference"
#define CONFCITY "Monterey"
#define ENGLISH 1
#endif
```

File g_config_people

```
#if SMITH
#define NAME "Smith james"
#define ADDRESS "London"
#define NUMBER 1
#define GENDER 1
#define MARRIED 0
#endif
#if FAVRE
#define NAME "Favre Jean-Marie"
#define ADDRESS "Grenoble"
#define NUMBER 1
#define GENDER 1
#define MARRIED 0
#endif
```

File g_feudal_title

```
#if GENDER == 1
"sieur"
#elif GENDER==0 && MARRIED==0
"demoiselle"
#elif GENDER==0 && MARRIED==1
"dame"
#endif
```

File g_plural

```
#if PLURAL
"s"
#endif
```

occurrence on the fourth line expands into "A+2", which in turn, expands into "10+2".

Even if the above sequence seems artificial, such problems tend to arise in practice when definitions are spread among the code and/or pertain to different include files. Is this strange behavior used in the g_letter system?

P3. Files refer to each other by means of file inclusions. Their relative order is relevant for the reader since each file can contain macro definitions. This is especially clear in the g_letter system. Where should we start reading?

P4. When there are a lot of files in a directory, it is not easy to get a clear idea of their possible interactions. With only seven files, it is difficult to understand the g_letter system as a whole. Are there any independent subsystems?

P5. It can be difficult to know which are the parameters of a CPP file. For instance, is it useful to set macro GROUPNB when using g_letter? Is it compulsory when using g_title?

P6. Removing variants is useful when a platform has no longer to be supported (for instance if it becomes obsolete or if the corresponding code is considered unmaintainable). Unfortunately, sharing code proves to be most awkward in performing such operations. For instance, let us assume that the g_letter system is now intended to be used only for international conferences, or that the maintainer is not able to deal with French idiomatic irregularities. Clearly, it would be better to remove French variants, but this is not an easy task. Changing or removing a line may cause side effects. In practice, variants are added but never deleted. This increases complexity, generates dead code, etc.

P7. The presence of numerous #if directives makes the structure almost impossible to follow. Often, human readers are not able to take into account all the variants at a time; they go over the same piece of code repeatedly, trying to understand only a few cases each time. This phenomenon is even more clear when a variant subset is more familiar to the reader: his first partial reading is aim to determine the purpose of the code, then he goes over the code and tries to understand other variants by analogy. For instance, an english speaking maintainer will naturally start to read the g_letter system focusing on English parts.
Whatever their reasons, maintainers try to understand CPP files by means of repeated partial readings. This process is very painful.

P8. Conversely, sometimes the reader has to focus his attention directly on a small portion of the code (for instance if a bug have been located at this place). If a modification is need, he ought to rapidly acquire a very precise knowledge of this piece of code. If this part contains macro occurrences or is enclosed in conditional directives, a local knowledge is not sufficient. A cautious maintainer has to examine all the files to determine if they contain instructions which can change the behavior of the piece of code. In practice, this is impossible with large systems. Maintainers are reduced to making assumptions based on their global knowledge; they scan a few files and deduce possible behavior. This results in a "hit or miss" maintenance process. Let consider for example, that a maintainer has to understand the last line of the g_letter file ("CONFCITY"). Reading only this file is not sufficient. It neither indicates how this macro is computed nor in what conditions this line is included in the letter. Similarly, who is able to make a safe modification of the g_plural file (only 3 lines) without uncontrolled side effects?

Note that most of the above problems are not specific to CPP and occur with other preprocessors as well. Put together, all of them can convert the maintenance of complex preprocessor files into a nightmare. The worst of it, is that such tedious tasks are not supported by any reliable tools. This can be explained in part by the fact that these problems are not well understood and seem very specific. Thus it is difficult to provide coherent and general solutions.

4. CPP from an abstract point of view

The aim of software engineering is to apply scientific knowledge to solve industrial problems. This term does not exclude the study of the most pragmatic problems; it only implies using abstractions.

4.1. APP: an abstract preprocessor

It is really difficult to think in CPP terms. Thus we have defined APP, an **A**bstract **P**re**P**rocessor.

The APP language is an abstract language. It has an abstract syntax but no concrete syntax. Indeed APP is used only to describe abstractions for CPP files. *This language is equivalent to CPP* but it is based on traditional programming-in-the-small concepts and terminology (variables, instructions, procedures, etc.). APP provides a convenient way to think about preprocessor files at an abstract level.

The APP language is very simple. It deals only with string variables and has only a few statements: a nop statement (no operation), an assignment, a conditional statement, an output statement, a procedure call, and a sequence of statements. A program is a list of named procedures. Below a simplified APP abstract syntax is given using few keywords to make the meaning of each construction more intuitive.

Syntactic domains		Abstract syntax
c ∈	Const	t ::= [c | **$**v]
v ∈	Var	s ::= **NOP** | v **:=** **"**t**"**
t ∈	Term	| **IF** t **THEN** s **ELSE** s
s ∈	Stmt	| **OUT** t | **CALL** pn | s **;** s
pn ∈	ProcName	p ::= **PROCEDURE** pn **IS BEGIN** s **END**
p ∈	Proc	pg ::= **PROGRAM** [p]
pg ∈	Prog	

Constants, variables and procedure names are atomic values. Terms, statements, procedures and programs are constructed. Only two constructors are used: the "|" symbol represents alternatives while square brackets are used for the list constructor.

4.2. From CPP to APP

Since APP describes abstractions for CPP files, each CPP entity correspond to an APP entity. Furthermore, the translation from CPP to APP can be fully automated. In this section the principle of this conversion is shortly described.

Basically, abstractions for CPP macros are APP variables and CPP directives correspond to APP instructions.

Clearly, macro definitions are variable assignments and

conditional compilation directives are conditional statements. It is more interesting to note that each text fragment included in a CPP file translates to an output statement taking as its argument an APP term. To put it more precisely, all macro names correspond to APP variable occurrences and all other tokens are converted to APP constants.

Since a CPP file is a sequence of CPP directives, 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![1]

At first sight, this abstraction schema is not intuitive. Perhaps, it is better illustrated by the following example which present a partial translation of the g_letter system.

```
PROGRAM
    PROCEDURE g_config_conference IS ...
       BEGIN
          IF $SMITH THEN
             NAME := “Smith james”
             ADDRESS := “London”
          ....
    PROCEDURE g_config_people IS ...
    PROCEDURE g_feudal_title IS ...
    PROCEDURE g_possessive IS ...
    PROCEDURE g_termination IS ...
    PROCEDURE g_title IS ...
    PROCEDURE g_letter IS
       BEGIN
          CALL g_config_people
          CALL g_config_conference
          OUT $NAME , $ADDRESS
       IF $ENGLISH THEN OUT “Dear”   ELSE NOP
       CALL g_title
       OUT $NAME
       IF $ENGLISH THEN
             OUT “We are pleased to be able to tell you”
             OUT “that the referees have accepted your”
             OUT “paper entitled “ $TITLE “submitted”
          ...
```

4.3. APP Denotational semantics

Speaking in APP terms is useful because it makes it possible to refer to programming-in-the-small knowledge. Nevertheless, changing words and syntax is not sufficient. The design of safe automated tools requires precise language semantics. To this end, APP denotational semantics have been formally defined in [1]. The complete definition takes about six pages and goes beyond the scope of this paper. Introducing only the main semantic function is sufficient here.

Semantic domains

$\varphi \in$ *Fragment* $\qquad \rho \in$ *Mem* = Var -> Term

Semantic functions

SemStmt $\in$ Stmt -> *Mem* -> (*Fragments* $\times$ *Mem*)

In simple terms, the semantic of a statement (*SemStmt*) is a function which takes a memory state (*Mem*), and yields a fragment (*Fragment*) and a new memory state. In other words, a statement can change the memory (if it contains an assignment) and can generate an output. The memory itself is a map which gives the terms associated with each variable (Var -> Term).

4.4. APP main characteristics

Writing down the formal semantics of APP might seem to be a purely academic exercise. This is not the case. Since APP is equivalent to CPP, the purpose is indeed to describe precisely the behavior of CPP, which is sometimes confusing or rather strange (see for example ***P1*** or ***P2***). We believe that this rigor is the only way to design reliable tools. In the context of this paper, only the main characteristic of APP can be presented.

Dynamic binding. While nearly all traditional programming languages use static binding, APP is based on dynamic binding. In the first class of languages, when defining a function, each occurrence of a variable is statically bound in the scope of the declaration. As ***P2*** points out, this is not the case for CPP.

This characteristic evidently appears in the description of APP semantics: while memory usually stores semantic values, APP memory (Mem) stores syntactic values (Term). In other words, the right hand side of an assignment is not evaluated; it is stored as such. Moreover, the evaluation of a variable occurence can lead to recursive evaluations if the value stored in the variable contains variable occurences. The next sections show that dynamic binding plays a major role in the design of analysis techniques for APP programs.

Global variables, no declaration, no data types. APP procedures have neither parameters, nor local variables. In other words, all data exchanges rely on global variable assignments. Obviously, with no data encapsulation at all, side effects naturally occur. The lack of procedure signatures leads to ***P6***. This problem is even more acute since the APP language requires no variable declaration at all (even for global variables).

Straightforward code. APP control statements are very simple. There are neither loops, nor goto statements. As a consequence control flow is especially simple. This characteristic can greatly simplify analysis techniques.

Summing up, APP is really primitive compared to programming-in-the-small languages. Indeed, as a

1. Moreover, since a directory is a set of CPP files, the corresponding abstraction is a module (i.e. a set of procedures), parametrized macros correspond to function with no side effect, etc. For simplicity, this paper present only basic features.

programming-in-the-large language, it has not been designed to describe algorithm nor manage data structures. It deals with variation problems.

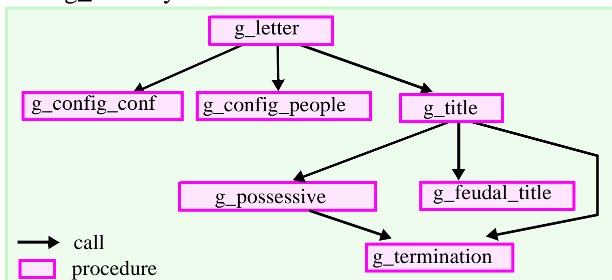
5. Application samples

Abstraction is useful to make explicit the characteristics of a subject. It also promote the reuse of existing solutions developed for subjects sharing the same abstraction. Since a set of preprocessor files can be seen as a program, all the techniques developed in computer science to manage and analyze programs are potentially applicable here! In this section, a few applications are presented.

5.1. Control flow analysis

The control flow of a program is a simple but useful structure. It can be used for example to compute complexity metrics (number of paths, Mac Cabe, etc.). This makes possible to trigger a restructuring plan when the complexity due to patches reaches an unacceptable level.

While control flow is defined at the statement level, in a call graph, nodes are procedures. In CPP terms, this concept correspond to the traditional “dependence relation”. It gives solutions to ***P3*** (Where to start reading?) and ***P4*** (Are there any independent subsystems?). Let consider the call graph of the g_letter system.



This graph provides valuable information to maintainers. Clearly, the g_letter file corresponds to the main system and reading should start from there. The g_title appears as an independent subsystem. To reduce system complexity, it can be wise to put g_possessive, g_feudal_title, g_termination and g_title in a specific directory (i.e. in a module in APP terms).

5.2. Data flow analysis

In a language like APP, data flow analysis is also of fundamental importance. As pointed out in section 4.4, all data exchanges are based on global variable assignments, giving rise to serious problems when no control is made. Data flow analysis can track suspicious information flow between unrelated procedures (this is apparent when call graph and data flow graphs do not overlap).

Interprocedural data flow analysis can also give answers to ***P5*** (which are the parameter of a file?). For that, we have designed a signature generation technique based on traditional computation like reaching definitions or live variables [22]. The problem was to adapt theses techniques to APP specificities, especially to dynamic binding. The APP abstraction has proven to be very useful: only a few inductive equations based on APP abstract syntax are necessary to generate signatures like the following.

```
PROCEDURE g_title (
   ENGLISH, GENDER, MARRIED : in ;
   NUMBER: in out;
   GROUPNB, SUBJECTNB, OBJGENDER, OBJNB, PLURAL : out)
```

The distinction is made between *in* parameters and *out* parameters. The value of *in* parameters are potentially used in the procedure, while the value of *out* parameters can be set or can changed by the procedure. These two classes are not exclusive; *in-out* parameters often correspond to parameters with a default value (if the parameter is not defined a default value is assigned to it).

With the above signature, the maintainer can easily guess that the title of a person is parametrized by the number of persons, the language used, the persons gender and their civil status. He can also suppose that a default value is available for the NUMBER parameter (this is an in-out parameter). Browsing the procedure code rapidly confirm this hypothesis. Finally, he can notice that this procedure changes the variables GROUPNB, SUBJECTNB, OBJGENDER, OBJNB. More complete data flow analysis shows that the values of these variables are not used outside this procedure. This characteristic usually indicates that they would be local variables if the language would allow this feature. This information provides advice to the maintainer: to avoid side-effects, these variables should not be used in other parts of the program.

Automatic signature generation provides more information that a reader can deduce by itself. For example, who is able to deduce the following signature for the g_letter system?

```
PROCEDURE g_letter (
  SMITH, FAVRE, DURHAM95, ICSM95,  TITLE: in ;
  NAME, ADDRESS, CONFNAME, CONFCITY,   NUMBER,
  GENDER, MARRIED, ENGLISH : in out ;
  GROUPNB, SUBJECTNB, OBJGENDER, OBJNB, PLURAL: out )
```

5.3. Program slicing

A program dependency graph (PDG) is a structure combining control flow and data flow to represent dependencies between statements [23]. Unrelated statements are not connected, even if they appear in sequence in the program. In fact, their respective order is irrelevant for the semantic of the program and PDGs capture that. The production of PDGs has been largely

studied in the last years. To produce APP PDGs, we have extended traditional algorithms to take into account dynamic binding.

Computing program slices is one of the main uses of PDGs. This notion is useful here since problem ***P8*** can be reduced to a backward slicing problem: which is the minimal instruction subset that should be read to understand the behavior of a specific piece of code? This problem, could be solved by a PDG traversal based on all statements of the piece of code.

Below a backward slice is computed for the last line of the g_letter file (see ***P8***). This program is semantically equivalent to the whole g_letter system with respect to the behavior of the selected line. This means that to understand it, the maintainer has to read only 7 lines instead of 161 lines in 7 files! Note that these ratios are even higher in practice: usually there is only few preprocessors directives, but they are spread into many files of hundreds or thousands lines.

```
IF $DURHAM95
  CONFCITY := ““Durham””
ELSE NOP
IF $ICSM96
  CONFCITY := ““Monterey””
ELSE NOP
OUT $CONFCITY
```

5.4. Program specialization

Reformulating ***P6*** and ***P7*** shows that they are closely related: ***P6***- How to definitively remove a variant subset to simplify the system? ***P7***- How to remove temporarily a variant subset to simplify the reading. Indeed, program specialization solves both problems.

Program specialization is a concrete operation defined on programs. Semantically, it corresponds to the domain restriction operation (the ◁ operator in VDM or Z specification languages). Given a constraint on possible parameter values, the goal is to produce a reduced version of the program which is semantically equivalent when applied to this domain.

Partial evaluation [24] and symbolic execution [25-27] are two possible techniques. In the former case, the constraint is expressed in terms of binding between a subset of variable and constant values. In the latter case, the constraint is defined using a general predicate, possibly using symbolic values. Symbolic execution, while being much more general, is usually not applicable to real programs.

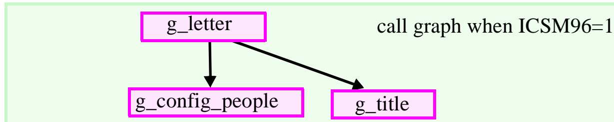
We have designed and implemented an intermediate technique taking into account APP specificities, dynamic binding included. A rigorous definition of this transformation has been elaborated from the APP denotational semantic (in the spirit of [27]).

Like other APP techniques, APP program specialization can solve practical problems (of ***P6*** and ***P7*** kind). Let us suppose that the user wants to specialize the g_letter system for the International Conference on Software Maintenance. Using signature generation, he deduce that ICSM96 is the corresponding variable. With, the constraint “ICSM96=1”, the program specializer reduces the g_letter procedure as shown in the next figure.

```
PROCEDURE g_letter (
  SMITH, FAVRE, TITLE: in ;
  NAME, ADDRESS, NUMBER GENDER, MARRIED : in out)
IS
BEGIN
   CALL g_config_people
   OUT $NAME , $ADDRESS
   OUT “Dear”
   CALL g_title
   OUT $NAME
   OUT “We are pleased to be able to tell you”
   OUT “that the referees have accepted your”
   OUT “paper entitled “ $TITLE “submitted to”
   OUT “the ICSM’96 conference.”
   OUT “We are looking forward to seeing you in “
   OUT “Monterey”
END
```

Note also that the procedure call to g_config_conference becomes useless. Since the conference working language is English, only the corresponding parts are selected (this is due to the ENGLISH:=“1“ assignment in g_config_conference). Finally, the conference name and location expand directly in the main text.

Program specialization can be used in conjunction with others techniques. The signature of the above specialized procedure is based on signature generation. Similarly, the specialized call graph of the system is as follow.



Compared to the original call graph, this specialization is really simpler. The conference configuration file naturally disappears, but this is also the case for g_possessive, g_termination and g_feudal_title. In fact these files were introduced into the system when it was originally developed for French letters.

As suggest in ***P6,*** if the maintainer is not able to deal with French idiomatic irregularities, it would be better to remove French variants from the system. APP program specialization makes this task fully automatic.

On the contrary, if the maintainer’s objective is a partial reading of all components (problem ***P7***), he may prefer to have a fully expanded view of the specialized system. In this case, procedure calls expand to the appropriate code taking into account each calling context. The next figure presents such a view. This procedure is semantically equivalent to the whole system assuming that the conference is ICSM’96. It seems rather complex, but indeed it gives an excellent view of what system do, without being too specific or too general.

Note that it is impossible to obtain such results with an APP interpretor (i.e. the CPP preprocessor): the output of such a tool is always a value, not a program; all preprocessors directives are removed and all macro occurences are expanded.

```
PROCEDURE g_letter (
  SMITH, FAVRE, TITLE: in ;
  NAME, ADDRESS, NUMBER GENDER, MARRIED : in out)
IS
BEGIN
   IF $FAVRE THEN
     NAME := "“Favre Jean-Marie”"
     ADDRESS := "“Grenoble”"
     NUMBER := “1”
     GENDER := “1”
     MARRIED := “0”
   ELSE NOP
   IF $SMITH THEN
     NAME := "“Smith james”"
     ADDRESS := "“London”"
     NUMBER := “1”
     GENDER := “1”
     MARRIED := “0”
   ELSE NOP
   OUT ““
   OUT $NAME , $ADDRESS
   OUT “Dear”
   IF ! $defined $NUMBER THEN
     NUMBER := “1”
   ELSE NOP
   IF $GENDER == 1 THEN
     IF $NUMBER > 1 THEN
       OUT “messrs”
     ELSE
       OUT “mr”
   ELSE
     IF $GENDER == 0 THEN
       IF $MARRIED == 1 THEN
         IF $NUMBER > 1 THEN
           OUT “misses”
         ELSE
           OUT “miss”
       ELSE
         OUT “mrs”
     ELSE NOP
   OUT $NAME
   OUT “We are pleased to be able to tell you”
   OUT “that the referees have accepted your”
   OUT “paper entitled “ $TITLE “submitted to”
   OUT “the ICSM’96 conference.”
   OUT “We are looking forward to seeing you in“
   OUT “Monterey”
END
```

Further specialization can be applied to the result of a specialization. For instance, if the maintainer wants to see what the letter looks like for a specific person, say Favre, he can add “FAVRE=1” to the specialization constraint.

```
PROCEDURE g_letter( TITLE : in ) IS
  BEGIN
    OUT “Favre Jean-Marie” , “Grenoble”
    OUT “Dear”
    OUT “mr”
    OUT “Favre Jean-Marie”
    OUT “We are pleased to be able to tell you”
    OUT “that the referees have accepted your”
    OUT “paper entitled “ $TITLE “submitted to”
    OUT “the ICSM’96 conference.”
    OUT “We are looking forward to seeing you in “
    OUT “Monterey”
END
```

6. The Champollion/APP environment

This work has been carried out in the framework of the Champollion project. The goal of this project is to study issues related to reverse-engineering-in-the-large [18,28], that is to say the intersection between Programming-in-the-large and Reverse-engineering. Current work is centered around preprocessors, shell scripts and makefiles reverse engineering.

All of the above techniques have been implemented in a prototype environment called Champollion/APP. Given a file system structure containing CPP files, this environment produces graphical and textual views upon request. The Champollion/APP architecture is classical for a reverse engineering environment. A component implements the abstraction schema from CPP to APP (section 4.2). It parses CPP files and generates APP Abstract Syntax Trees (section 4.1). ASTs are the central representation for APP programs. An APP interpretor is provided. This interpreter is used to check the CPP/APP equivalence and performs a few dynamic analyses not described in this paper. Various other components are in charge of generating different kinds of graphs useful for static analysis: control flow graphs and call graphs (section 5.1) program dependency graph (section 5.3), etc. These graphs are in turn used to compute a few metrics, procedure signatures as well as slices and specialized procedures. Finally, different components transform all this information into textual or graphical views managed by a text editor and a graph editor.

In concrete terms, emacs is used for textual views, while daVinci is used for graphical ones. The CPP to APP translation is based on Lex and Yacc. Other components have been developed using an object-oriented functional programming language (objective-caml, an ML dialect).

The picture on the last page shows a Champollion/APP session based on the g_letter system (a control flow graph, a program dependency graph and a slice are displayed). All the examples presented in this paper have been generated by our prototype.

Currently, this prototype suffers from some limitations. The CPP parser does not take into account the full lexical

conventions of the C language. Parametrized macros are not either recognized. These implementation restrictions prevent from applying this tool to most real-life C programs. These limitations should be removed in the future. Our goal is to experiment these techniques on large C software products like X-window or gun compilers.

7. Conclusion

The use of preprocessors is one of the more empirical activities involved in software production today. Few decades after their creation, they are still in use and they will remain so for a long time, even if they lead to unmaintainable structures. At first sight, problems seem very specific and are often unclear.

We have defined APP, an abstract language semantically equivalent to CPP and shown the usefulness of this abstraction. Classical techniques, usually applied in the Programming-in-the-small field, reveal to be good solutions to variation problems, a typical Programming-in-the-large issue. In fact preprocessors bridge the gap between PITL and PITS.

Contrary to [16,19], which focus only on conditional compilation, our approach takes into account the full complexity of preprocessor files. It is also more general. The APP language can be extended to deal with other control structure like loops, used for instance by the Compile Time Facility (CTF), the preprocessor associated to PL/1 language.

References

[1] J.M. Favre; “Une approche pour la maintenance et la ré-ingéniérie globale des logiciels”, Ph.D. dissertation, University of Grenoble, France, October 1995.

[2] A. Mahler; “Variants: Keeping Things Together and Telling Them Apart”, Chapter 3, in W.F. Tichy, Editor; “Configuration Management”, Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994.

[3] W. M. Gentleman and al. ; “Commercial Real-time Software Needs Different Configuration Management”, in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, October 1989, pp. 152-161.

[4] D. Tilbrook, R. Crook; “Large Scale Porting through Parametrization” in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 209-216.

[5] B.P. Munch; “Versioning in Software Engineering Database: the Change Oriented Way”, Ph.D. dissertation, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, 1993.

[6] P. Singleton, O.P. Brereton; “A Case for Declarative Programming-in-the-Large”, in Proc. 5th International Conference on Software Engineering and Knowledge Engineering, San Francisco, California, 1993.

[7] A. Zeller; “Configuration Management with Feature Logics”, Technical report TR-94-01, Technische Universitat Braunschweig, Germany, March 1994.

[8] Proc. of International Workshop on Software Configuration Management. 1st SVCC Grassau 1988, 2nd Princeton 1989, 3rd Trondheim 1991, 4th Baltimore 1993, 5th Toronto 1995.

[9] J.F.H. Winkler, C. Stoffel; “Program-Variations-in-the-Small”, in Proc. International Workshop on Software Version and Configuration Control, January, 1988, pp. 175-196.

[10] 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, May 1993, pp. 225-228.

[11] F.J. Grosch, G. Snelting; “Polymorphic Components for Monomorphic Languages” in Proc. of the 2nd International Workshop on Software Reusability, pages 47-55, Lucca, Italy, March 1993, pp. 47-55.

[12] 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.

[13] A. Litman; “An Implementation of Precompiled Headers” in Software - Practice and Experience, 23:3, March 1993, pp. 341-350.

[14] K.P. Vo, Y.F. Chen; “Incl: A Tool to Analyze Include Files” in USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 199-208.

[15] D. Spuler, A.S.M. Sajeev; “Static Detection of Preprocessor Macro Errors in C”, Technical report 92-7, James Crook University, 1992, 18 pages.

[16] M. Krone, G. Snelting; “On the Inference of Configuration Structures from Source Code”, Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994.

[17] P. E. Livadas, D.T. Small; “Understanding Code Containing Preprocessor Construct”, in IEEE Third Workshop on Program Comprehension, Washington, November 1994.

[18] 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.

[19] G. Snelting; “Reengineering of Configurations Based on Mathematical Concept Analysis”, CS report 94-02,Technical University of Braunschweig, Germany, January 1995.

[20] R. Stallman; “The C Preprocessor”, GNU Project, Free software foundation, July 1992, 52 pages.

[21] J.M. Favre; “The CPP Paradox”, 9th European Workshop on Software Maintenance, Durham, UK, September 1995.

[22] A.V. Aho, R. Sethi, J.D. Ullman; “Compilers: principles, techniques, and tools”, Addison-Wesley (1986).

[23] S. Horwitz, T. Reps; “The Use of Program Dependence Graphs in Software Engineering”, in Proc. 14th International Conference on Software Engineering, Melbourne, Australia, May 1992, pp. 392-411.

[24] N.D. Jones, C.K. Gomard, P. Sestoft; “Partial Evaluation and Automatic Program Generation”, Prentice Hall, ISBN 0-13-020249-5, 1993, 415 pages.

[25] J.C. King; “Program Reduction using Symbolic Execution” in ACM SIGSOFT Software Engineering Notes, Vol. 6, N. 1, January 1981, pp. 9-14.

[26] P.D. Coward, D.C. Ince; “The Role of Symbolic Execution in Software Maintenance” in Journal of Software Maintenance: Research and Practice, Vol. 3, 1991, pp. 183-192.

[27] A. Coen-Porisini, F. DePaoli, C. Ghezzi, D. Mandrioli; “Software Specialization Via Symbolic Execution” in IEEE Transactions on Software Engineering, Vol, 17, N. 9, September 1991, pp. 884-899

[28] J.M. Favre; "Reverse-Engineering and Configuration Management: Concepts and Perspectives", Software Conf'96, Paris, France, June 1996.