

TDK-Dolgozat

Daniel Dan

University of Szeged

Department of Software Engineering

Towards Better Variability Management in Industrial Software Systems

Scientific Students' Associations

*Author: Daniel Dan
Student
University of Szeged*

*Supervisor: Dr. Ferenc Rudolf
Assistant professor
University of Szeged, Department of Software Engineering*

*External supervisor: Dr. Stan Jarzabek
Associate professor
National University of Singapore, Department of Computer Science*

Table of Contents

1	Introduction	4
1.1	Background and problem description.....	4
1.2	Objective and scope of the work.....	5
1.3	Structure of the document.....	5
2	Background	6
2.1	Description of SPL	6
2.2	Variability management problem.....	7
2.3	XVCL – uniform variability technique	7
2.4	Clones and generic design	9
2.4.1	Generic design with XVCL	11
2.4.2	Benefits of generic representation of clones with XVCL.....	13
3	CPP	14
3.1	Cpp Overview	14
3.1.1	Inclusion of header files.....	15
3.1.2	Macro expansions.....	15
3.1.3	Conditional expansions.....	16
3.2	Limitations of variability management with cpp	17
4	CPP+	18
4.1	Motivation.....	18
4.2	Description.....	20
4.3	Usage examples.....	20
4.3.1	Adapting files.....	20
4.3.2	The break-insert mechanism	23
4.3.3	Cpp+ variables	25
4.3.4	Scoping rules.....	25
4.3.5	The while and select commands	27
4.3.6	Expressions	30
4.4	Evaluation	33
5	Linux Kernel case study.....	34
5.1	Introduction to Linux.....	34
5.2	Linux kernel and cpp	35
5.2.1	Kernel (of an operating system) in general.....	35
5.2.2	Linux kernel	37
5.2.3	How cpp is used in Linux kernel.....	37
5.3	Clones in Linux kernel	39
5.4	Sketch of solution with cpp+	41
5.5	Evaluation	45
6	Conclusion.....	45

1 Introduction

1.1 Background and problem description

A lot of software systems have more versions, and more communities using them in different ways. Because of cost and time constraints the developers must find a way to provide variability options of the software without implementing the whole system from scratch. There is always a set of *assets* of the system (*core assets*), which is present in all configurations. *Software Product Line* (SPL) approach focuses on domain-specific reuse, within a family of software systems that are known to have much in common with one another. The core idea of an SPL approach is to set up a base of reusable assets, so-called SPL Architecture. Specific products are then built by reusing assets stored in a SPL Architecture. We can find more details about SPL in section 2.1

Design of generic SPL core assets is a difficult task. We also need to handle product variant features that often have wide-spread impact on software components. We can do this task with *variability management techniques*. With these techniques we can add variation points to software, and with these variation points we can customize it. In this document we focus on variability management with *cpp* (C preprocessor), one of the most common variability management techniques. With *cpp* we can implement the variation configuration in easy way with preprocessor directives. However the program code instrumented with *cpp* macros can be hard to understand and change. In section 3.2 we describe the limitations of this variability management technique.

We can use as an example of a big software system the Linux operating system, which has many versions. All versions are based on a common code base. A required Linux version can be derived from the base by selecting required features (via configuration options) [1]. To facilitate such derivation of custom Linux systems, each feature is mapped into variation points in the Linux base at which feature-related code is included to or excluded from the base. Variation points are marked with pre-processing (*cpp*) commands (such as *#if*). The whole configuration process is controlled by scripts.

We can consider Linux system versions as a Software Product Line (SPL) [2][3]. Common code base is an SPL Architecture, *cpp* and scripts are variability techniques used to create

SPLA and to support derivation of custom Linux systems. Many industrial SPLs use pre-processor as one of the main variability technique.

Applying ad hoc techniques such as pre-processing and scripts for variability management in SPL becomes complicated in case of many variant features and feature dependencies. [4][5].

1.2 Objective and scope of the work

Better engineering qualities can be achieved with variability techniques designed especially for variability management in SPLs. An example is Frame Technology [6] that yielded excellent results in large industrial SPLs. XVCL implemented at NUS is based on frame concepts [7]. XVCL can do what cpp and scripts can and much more, leading to elegant, easy to work ways to represent program with variants (such as Linux base with variant features). XVCL Processor automates the process of deriving custom systems from SPLA.

XVCL is a dialect of XML. Therefore, each source file must be converted to XML file before processing. In this document we present *cpp+*. With *cpp+* we can use XVCL processing rules in notation free of XML syntax, therefore it will be easier to understand and apply. Cpp+ has some additional features compared to XVCL, and provides a flexible syntax, which can be redefined by the user. In this document I represent an approach to make configuration and understanding of a software system easier and more elegant.

1.3 Structure of the document

The structure of present document is the following:

- In first chapter (this chapter) I describe the background of the work, the main problem, how we try to solve this problem.
- In the second chapter we can read more detailed information about the background of the work. I will describe the Software Product Lines (SPL), the main problems of variability management, how mostly is used, and a tool implemented at NUS National University of Singapore (NUS) the XML-based Variant Configuration Language (XVCL).
- In third chapter we can read about a common *macro processor* used for variability management: the C preprocessor (cpp). We can read about how it is working, how we can use, and what the limitations of variability management with cpp are.

- In the fourth chapter I will describe the cpp+, how is working, how we can use it, and some simple usage examples.
- In the fifth chapter we can read about the Linux case study. We used cpp+ to make the representation of 9 files in the Linux kernel code. We describe the process, the results, and its advantages.
- The last chapter will be the conclusion of the work.

2 Background

2.1 Description of SPL

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be increased.

Reuse is based on the premise that much similarity exists in software systems. This is particularly true for systems in a given application domain such as customer order processing or finances. Other examples include gear control software for different car models or versions of Linux for different computers. Systems in each of those application domains are similar one to another. They will also differ in features that are specific to a given context of system usage.

Software Product Line (SPL) [17] comprises a family of similar software systems, built by adaptive reuse of shared *SPL core assets*. Core assets include all software artifacts such as architecture, code components, documentation, test cases, etc.

SPL core assets are reused (after possible adaptations) to derive custom systems from them. This system derivation process is a core idea of SPL, as it is the source of expected productivity improvements in development and maintenance of systems that form an SPL. Therefore, system derivation should be highly automated. SPL community is in the quest for

variability management techniques capable of formalizing, streamlining and automating the custom system derivation process.

2.2 Variability management problem

Design of generic SPL core assets is the main technical challenge in SPL engineering. Conventional design of software architecture and components is a prerequisite to reuse, but is not sufficient. In addition, we need to handle product variant features that often have wide-spread impact on software components. Variability management techniques are to address this challenge.

Variability techniques allow us to add variation points to software at which software can be customized during reuse. If pre-processing such as `cpp` is used to manage variability, then variation points are marked with macros `#if` or `#ifdef`.

Conventional variability techniques (such as pre-processing, configuration parameters, scripts, build tools, and software configuration management systems) do not offer scalable solutions to variability management problems. Conventional variability techniques were not designed for the purpose of supporting SPLs. Each such technique handles one variability problem, but fails to handle others. Therefore, it is a common SPL practice to apply many conventional variability techniques to build SPL core assets. However, applying many incompatible variability techniques creates yet other problems: It leads to manual, error-prone and time-consuming product derivation, hampering reuse productivity. Ad hoc variability techniques poorly scale to SPLs with many variant features and complex feature dependencies.

Researchers venture into unconventional meta-level variability management techniques, created just for the purpose of variability management [31][32]. XVCL (described in Section 2.3) is one of such techniques.

2.3 XVCL – uniform variability technique

XVCL (XML-based Variant Configuration Language), developed in Software Engineering Lab at NUS, is an open source and public domain variability technique and tool. Initially, this work focused on variability management, to help domain engineers design generic, adaptable core assets, and to automate reuse-based product development. It is a uniform technique to

manage variability in SPL core assets such as code, architecture, documentation (e.g., written in WORD), models, test cases, etc.

XVCL has been applied to manage variability in SPLs in a range of application domains (command & control systems, business applications, games for mobile phones, Web portals), using variety of programming technologies (Java, ASP, .NET, JEE).

XVCL is a uniform variability technique that can do more than conventional variability techniques, easier, with less errors during product derivation, and faster (as its application is automated by XVCL Processor). What makes XVCL compelling as compared to other variability techniques is its practicality and scalability. Any conceivable pattern of similarities and differences in software applications can be captured in XVCL representation. XVCL method scales to large systems. Results to date suggest that the bigger the products and the more they vary, the higher the productivity gains, as compared to conventional reuse [6][33][34]. It is demonstrated in lab studies and small industrial projects that XVCL has the potential to raise productivity levels beyond what is possible with conventional approaches, and that XVCL can manage similarities/differences situations [35] which other methods developed in research cannot manage [18][36].

A meta-component in XVCL is an XML file with program code (written in any programming language), instrumented with XVCL commands. Using XVCL, we organize meta-components into a layered hierarchy called an x-framework. Meta-components at lower-levels are building blocks for the higher-level meta-components - XVCL processor composes higher level meta-components from lower-level meta-components, after applying possible adaptations to them. An x-framework forms a base of reusable assets, such as product line architecture, from which we build custom systems.

XVCL has strict rules for its syntax. Is a dialect of XML, so the usual XML rules apply to XVCL. In addition, well-formed x-frames must obey rules that are specific to XVCL. In the next example we can see an example x-frame, with some simple XCVL commands:

```
<?xml version = "1.0"?>
<!DOCTYPE x-frame SYSTEM "DTDlocation">
<x-frame name="A">
    <set var="y" value="YB1"/>
    <set var="x" value="XB2"/>
    <adapt x-frame="B.xvcl">
        <insert break="breakX">
            xAB
```

</x-frame>
</adapt>
</insert>

2.4 Clones and generic design

Clones are program structures repeated many times in a software system in variant forms. These repetitions range from similar code fragments, so-called simple clones, to large-granular structures such as files or directories, called structural clones [11].

Studies show that even in well-designed programs, we typically find 50%-90% of code contained in clones, repeated many times. For example, the extent of the redundant code in the Java Buffer library was 68% [10], in parts of STL (C++) - over 50% [11], in J2EE Web Portals – 61% [37], and in certain ASP Web portal modules – up to 90% [34].

Generic design techniques aim at non-redundancy in software. Generic design can replace clones with generic representations. We can achieve simplifications of a software system proportional to the frequency of clone occurrence, which are often proportional to the size of a software system. C++ templates and Java generics exemplify such generic representations. STL, a library of C++ data structure classes is a classic example of simplifications that can be achieved by avoiding repetitions. However, type-parameterization (called templates in C++) and other techniques such as *iterators* used to build generic data structures and algorithms are not sufficient to achieve genericity in other domains [37][10] (even in STL some of the repetitions remain unresolved [11]).

The engineering benefits of non-redundancy become evident especially if we pay attention to large granularity clones.

Suppose we have 10 user interface forms $a_1, \dots, a_{10}$ (e.g., data entry forms). Each form, say a_i , interacts with 5 business logic functions b_i, c_i, d_i, e_i, f_i (e.g., data validation rules or actions to be performed upon data entry at certain fields). We further assume that there is considerable similarity among the 10 forms, and business logic functions that each form needs. (You can think about user interface forms and business logic functions as any components, subsystems or interacting systems.)

If each form and each business logic function is implemented as a separate component, we have to manage $10 + 10 \cdot 5 = 60$ components and 50 interactions (situation shown in Figure 2.1 (a)).

Suppose we unify similar business logic functions with five generic business functions B, C, D, E, and F. We reduce the solution space to $10 + 5 = 15$ components and 50 interactions, with the slightly added complexity of a generic representation. The interactions have become easier to understand, as each form interacts with components that have been merged into five groups rather than 50 distinct components (situation shown in Figure 2.1 (b)).

Suppose we further unify 10 user interface forms with one generic form A. We reduce the solution space to $1 + 5 = 6$ components, with five groups of interactions, each group consisting of 10 specific interactions, plus the complexity of a generic representation (situation shown in Figure 2.1 (c)).

Software validation and changes performed at the level of the generic components would be easier than at the level of concrete components due to smaller number of distinct components and interfaces that have to be analyzed and clearer visibility of change impact. Generic software structures also form natural units of reuse, potential building blocks of other similar systems (e.g., forming Product Lines).

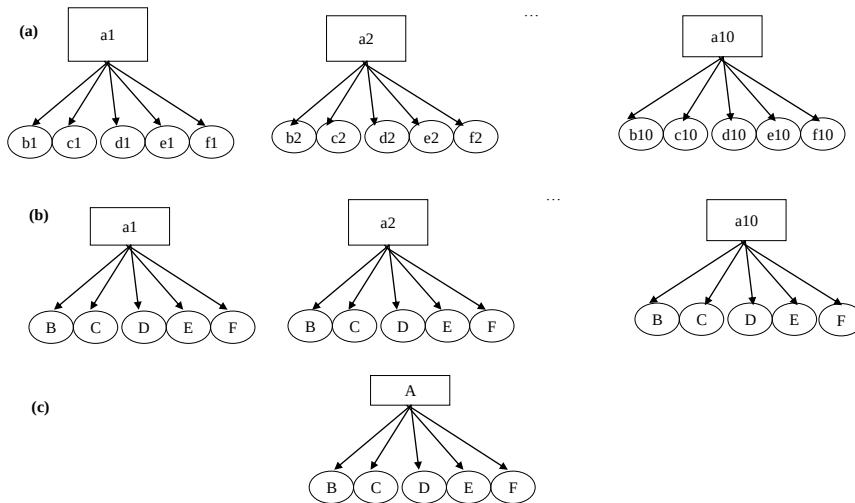


Figure 2.1. The impact of unifying similarities with generic structures on complexity

The above situation has been abstracted from a Domain Entity Management Subsystem (DEMS) of a command-and-control application developed in C# by the XVCL team's industry partner ST Electronics Pte Ltd (STEE). DEMS involves domain entities such as *User*, *Task* or *Resource*. For each entity, there are operations, such as *Create*, *Update*, *View*, *Delete*, *Find* or *Copy*. STEE's intention was to make DEMS a part of the Product Line architecture, so DEMS was meant to be generic, adaptable and extensible.

The design of each operation such as *CreateUser* or *CreateTask* involves a pattern of collaborating classes from GUI, service and database layers. Each box in Figure 2.2 represents a number of classes: GUI classes implement various forms to display or enter data; business logic classes, implement data validation and actions specific to various operations and/or entities. Entity classes define data access, classes at the bottom contain table definitions.

Classes in corresponding boxes at each level display much similarity, but there are also differences induced by different semantics of domain entities: For example, operation *CreateTask* requires different types of data entry and data validation than *CreateUser*.

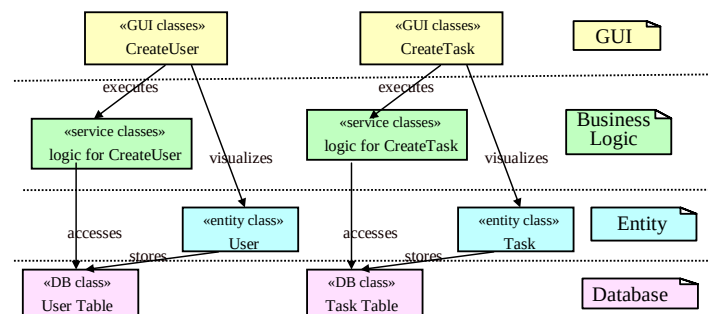


Figure 2.2. A recurring pattern of components

Similarities among operations are so substantial that it is tempting to view operations as instances of a generic operation. C# language features (including generics proposed for C#) and .NET mechanisms were first considered to define such generic solutions. However, the nature of variations across operations was such that it was not possible to design a generic solution for groups of similar operations (e.g., *Create[entity]Form* or *Update[entity]Form*). Therefore, implementation of operations for different entities was replicated many times, with required variations. DEMS comprised 18,823 LOC of C# code, with 117 classes covering GUI, service and database layers.

2.4.1 Generic design with XVCL

The number of variation points in software is proportional to the size of a software system. Generic design reduces the size of a software system. At the generic representation level, we deal with much a smaller number of variation points.

While repetitions are often unavoidable and intentional in executable programs, they can be avoided (unified) with unconventional generic representations. This is analogous to Aspect-

Oriented Programming [32] where a *meta-level extra plane* is created to modularize concerns that otherwise inevitably crosscut conventional program modules (classes).

The above observations motivate the XVCL approach. Developers write the functional core of their program solutions in programming languages of their choice. They use XVCL to organize code in-the-large to achieve non-redundancy, ease of changing, reusing and versioning.

We can separate XVCL representations to four levels. At first level represents each significant group of similar program structures in a unique, generic, but adaptable form. In second delineates the differences among specific program structures in each group as deltas from their generic form. In third level records the exact location of each such structure in a program. And in the fourth level, automates deriving specific structures from their generic forms to produce an executable program.

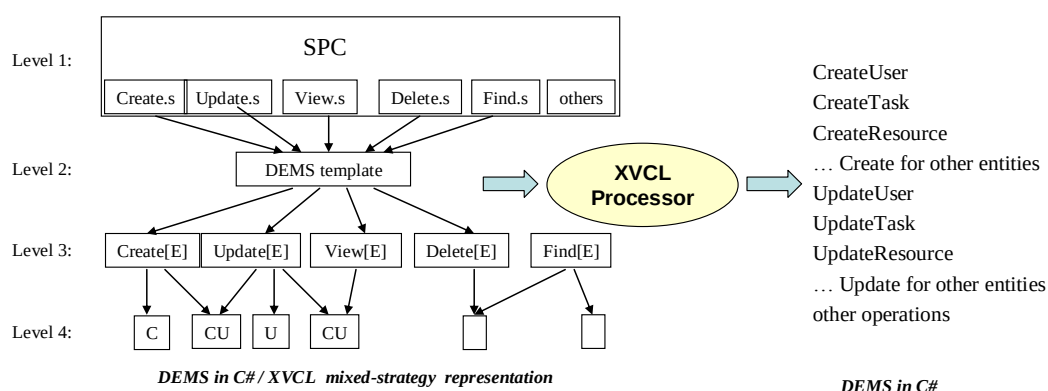


Figure 2.3. Hierarchical unification of similarities

Figure 2.3 shows an outline of non-redundant DEMS in C#/XVCL software representation. Boxes represent solution's building blocks called frames [6]. At Level 3, each group of operations such as *CreateUser*, *CreateTask*, ... has been represented by one generic operation parameterized by a respective domain entity. Similarities among different operations for the same entity (e.g., *CreateUser*, *UpdateUser*, ...) are unified at Level 4. Frames at Level 4 represent generic classes, building blocks for DEMS operations, as indicated by frames referenced from more than one operation (e.g., generic classes labeled with CU are reused in construction of *Create* and *Update* for various entities). The top-most frame, called an SPC, contains global controls and parameter settings that specify the overall process of constructing DEMS from frames below. "DEMS template" at Level 2 defines the structure of DEMS

architecture, that is the organization of component patterns implementing various operations plus any other functions supported by DEMS, not discussed in this example.

The XVCL Processor interprets the framed representation to synthesize DEMS in C#.

In this sketchy solution outline, we do not distinguish among classes belonging to the four system layers.

In most systems, software structures such as in Figure 2.3 are implicit, and must be expressed in the form of external documentation. XVCL makes such views explicit and an integral part of a formal program representation. Frames contain both code and the knowledge of program design, integrated into one representation. For example, the C#/XVCL framed software representation contains C# code needed to produce all the DEMS operations, and also information helpful in maintenance/reuse, such as the record of similarities and differences among operations for different entities. In most systems, such information is implicit, and must be expressed in the form of external documentation. XVCL makes it an integral part of a formal program representation.

In this example, the size of the C#/XVCL representation is 68% smaller (in terms of lines of code, without blanks or comments) and conceptually simpler than its C# counterpart. Size reduction also leads to improved maintainability. Typical enhancements (such as change of business rules for domain entities or adding/deleting a new entity or operation) require less modifications with smaller impact in the C#/XVCL DEMS as compared to its C# counterpart. Finally, it is quite easy to reuse the C#/XVCL DEMS in other similar command and control systems.

2.4.2 Benefits of generic representation of clones with XVCL

To illustrate the impact of using a uniform variability technique to design SPL core assets on software productivity, XVCL team showed their experiences from a project in which ST Electronics (Info-Software Systems) Pte Ltd used XVCL to manage variability in an SPL of web portals. This project involved a family of Team Collaboration Portals (TCP) in ASP [34]. Initially, ST Electronics applied state-of-the-art design methods to maximize reusability among TCPs. Still, a number of problems were observed that could be improved by applying XVCL. The benefits of an XVCL TCP reuse framework after development of nine portals are illustrated in **Table 1**.

1	code size of a TCP	30,000 LOC
2	managed code in original TCP	16,322 LOC
3	managed code in TCP with XVCL	5,696 LOC (65% reduction)
4	effort to build the first-cut XVCL representation	two man/weeks to support three portals and observe first benefits
5	effort to build new portal with XVCL	10% of unique custom code required for new portal; an estimated eight-fold reduction of development effort and time-to-market
6	effort to maintain existing portals with XVCL	managed code for nine portals was 22% less than the size of the original single portal

Table 1. Productivity figures from industrial application of XVCL to web portals

3 CPP

3.1 Cpp Overview

The C preprocessor – often known as *cpp* – is a macro processor that is used automatically by the C compiler to transform the program source before compilation [13][14]. Although the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

C preprocessors vary in some details. In context of this work I use the GNU C preprocessor [13], which provides a small superset of the features of ISO Standard C. In this section I will introduce the mostly used practices and *cpp* directives.

The C preprocessor provides four main facilities. These are:

- Inclusion of header files
- Macro expansion
- Conditional expansion
- Line control (we do not deal with line control in this document)

3.1.1 Inclusion of header files

One of the most common uses of the preprocessor is to include another file. We can do that with the **#include** command. For example:

```
#include <stdio.h>
#include "Rectangle.h"
```

The preprocessor replaces the line of the *#include* statement with the text of the included file. If the filename is enclosed within angle brackets, the file is searched for in the standard compiler include paths. If the filename is enclosed within double quotes, the search path is expanded to include the current source directory. We must prevent the double inclusion of a file. We can do this using *#include guards* or with *#pragma once*.

3.1.2 Macro expansions

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. We can use two types of macros with cpp, *object-like* macros and *function-like* macros.

Object-like macros:

An *object-like* macro is a simple identifier which will be replaced by a code fragment. We use this to define constants in the code.

```
#define MAX = 10
```

The above object-like definition causes the preprocessor to replace all subsequent instances of the identifier MAX with the constant 10.

Function-like macros:

An identifier followed by a parameter list in parentheses and the replacement tokens. We can define function-like macros with the *#define* directive with a pair of parentheses immediately after the macro name. For example:

```
#define SUM(a,b) (a + b)
```

After this definition, the following statement

```
int sum = SUM(x,y);
```

will appear in the output of the preprocessor as:

```
int sum = (x + y);
```

We can use in macro expansions the `#` and the `##` operators. The `#` (single number sign) operator converts a parameter of a function-like macro into a character string literal. The `##` (double number sign) operator concatenates two tokens in a macro invocation.

The C preprocessor runs through the file top-to-bottom and treats `#define` statements like a copy-and-paste operation. Once it encounters the line `#define XY` command, it starts replacing every instance of the word `XY` with the given value. The pre-processor does not process (or even notice) C-language scoping mechanisms like parenthesis and curly braces. The scope of a macro definition begins at the definition and does not end until a corresponding `#undef` directive is encountered. If there is no corresponding `#undef` directive, the scope of the macro definition lasts until the end of the translation unit.

3.1.3 Conditional expansions

In a macro processor, a *conditional* is a directive that allows a part of the program to be ignored during compilation, on some conditions. These can be arithmetic expressions or testing whether a name is defined as a macro.

The reasons for using conditional expansions are the following:

- A program may need to use different code depending on the machine or operating system it is to run on.
- A program can be compiled in different ways. For example for debugging we can run the program in different way.
- We can use a condition which is always false to exclude a part of code from the program.

A conditional expansion can start with `#if`, `#ifdef` or `#ifndef`, can contain zero or more `#elif` directives, zero or one `#else` directive, and one matching `#endif` directive.

```
#ifdef VAR
...
#endif

#if X == 1
...
#elif X == 2
```

```

...
#else /* X != 2 and X != 1*/
...
#endif

```

The special operator “*defined*” is used in *#if* expressions to test whether a certain name is defined as a macro. *#ifdef* VAR is equivalent to *#if defined* VAR, and *#ifndef* VAR is equivalent to *#if ! defined* VAR.

3.2 Limitations of variability management with cpp

To implement variability in software product lines, developers often use conditional-compilation mechanisms of the C preprocessor cpp. With *#ifdef* and *#endif* directives they frame code fragments that are conditionally excluded during the compilation process [15][16].

There are well-known problems involved in managing large number of configuration options in base code with cpp. As the number of configuration options grows, programs instrumented with cpp macros become difficult to understand, test, maintain and reuse. It is difficult to figure out which code is related to which requirements, and to understand or change program in general. Managing configuration options with *#ifdefs* is technically feasible, but is error-prone and does not scale [5].

On design-level means is important to manage product variants in reuse-based way [17][18]. Mappings between features, reusable components and specific variation points in components affected by features are often complex. Problems magnify in the presence of feature dependencies, when the presence or absence of one feature affects the way other features are implemented [19]. Feature dependencies lead to overly complex conditions under *#if*, or many nesting levels under *#ifdef* macros. Fortunately, there are not complex feature-dependencies in Linux.

Since cpp can be used with different host languages, arbitrary changes such as coarse- and fine-grained extensions are possible. Coarse-grained extensions add new functions or data structures, whereas fine-grained extensions add source code pieces, such as statement and expression extensions or function signature changes.

Depending on the feature selection, which is provided as command line parameters or configuration file, the preprocessor generates different variants of the program, with or without these code fragments. Conditional compilation with the C preprocessor is a very

simple form of implementing compile-time variability, easy to learn and broadly used in practice.

Despite many benefits of architecture- and component-based approaches to reuse, managing features that have fine-grained impact on many reusable components requires extensive manual, error-prone customizations during product derivation. Unfortunately, the C preprocessor has several properties that render it error prone and difficult to analyze.

The following properties makes difficult to analyze code that was not already preprocessed (called unpreprocessed code or pre-cpp code) [15]:

- The C preprocessor is token-based and uses lexical macros [28] which is oblivious to the underlying language. While the preprocessor can be used language-independently, it can also easily introduce syntax errors, such as parenthesis mismatch, in the host language. The preprocessor has no mechanism to detect potential problems in the underlying code. Conversely, any non-heuristic syntactic or semantic analysis of code before preprocessing is even regarded to be impossible by some [29].
- The evaluation of conditional-compilation directives in the C preprocessor is deeply intertwined with the file-inclusion mechanism (*#include*) and macro facilities (*#define* and *#undef*). Using conditional compilation, alternative expansions of macros can be defined; and macro definitions can influence the evaluation of conditional-compilation directives.
- Conditional compilation is not only used to implement compile-time variability for features in the product-line sense or low-level portability issues, but also for include guards. An include guard is a pattern that prevents multiple inclusion of the same file, which uses the same conditional-compilation mechanisms and is difficult to distinguish from variability by tools.

4 CPP+

4.1 Motivation

Despite the above mentioned problems and weaknesses, cpp is often used for variability management. Besides component design preprocessors are often used to manage fine-grained

features in bigger systems. Poor readability, scattering of feature code across the base code and feature interactions makes the preprocessed and the unprocessed code hard to understand, test, maintain and reuse.

Large size of base components, large size of variation points and complex impact of features on base components is the source of problem with preprocessing-like approach to manage system variants. Using XVCL [7], the approach mentioned in section 2.3, can alleviate this problem. The core idea of XVCL is to indentify groups of similar base components and represent them as a generic, parameterized base component, likely to C++ templates and Java generics. The mentioned technologies can represent groups of similar classes differing in type parameters in generic form. XVCL can represent any group of similar source files or directories differing in arbitrary ways in generic form.

With cpp+ we planned an easier and simpler syntax, which is similar to cpp commands. The XVCL syntax is in a strong correlation with XML syntax. So the prerequisite of understanding and writing XVCL source is to know the XML syntax and rules. Cpp+ does not have this restriction. In addition cpp+ has some more features like XVCL, for example the so-called *set-loop*, which makes possible for the user to define a structure especially for loops in cpp+. Other changes are the flexible syntax (which provides the user to redefine cpp+ syntax), and the order of the cpp+ commands and nesting in each other (like *#select* in *#adapt*). On Figure 4.1 we can see the difference between XVCL and cpp+ syntax.

The figure displays two side-by-side code snippets. The left snippet is in XVCL syntax, using XML-like tags such as `<?xml version="1.0"?>`, `<!DOCTYPE x-frame SYSTEM "file:///c:\xvcl\dtd\xvcl.dtd">`, `<x-frame name="while" outdir="whileDir" outfile="while" language="java">`, `<set-multi var="loop" value="1,2,3"/>`, `<adapt x-frame="a.xvcl">`, `<insert break="1">`, `111`, `</insert>`, `<insert break="2">`, `222`, `</insert>`, `<insert break="3">`, `333`, `</insert>`, `</adapt>`, and `</x-frame>`. The right snippet is in cpp+ syntax, using commands like `#output resources/While/output`, `#set loop = 1,2,3`, `#adapt: resources/While/a`, `#insert 1`, `111`, `#endinsert`, `#insert 2`, `222`, `#endinsert`, `#insert 3`, `333`, `#endinsert`, and `#endadapt`.

Figure 4.1: The same code fragment in XVCL syntax (left) and cpp+ syntax (right)

With cpp+ we can improve the variability management with extended cpp syntax, and we can use the concepts provided by XVCL language.

4.2 Description

Cpp+ is a meta-programming technique like XVCL. With cpp+ we can use effective reuse mechanisms in a simple way. We can consider it as an upgraded C preprocessor implemented to make variability management of software systems easier to understand, change and maintain. Cpp+ uses technologies provided by XVCL (discussed earlier in section 2.3) for variability management.

With cpp+ we can apply XVCL concepts and mechanisms to manage variants in a range of product line assets such as software architecture, program code, test cases, technical and user-level program documentation or requirement specifications. With cpp+ we can use the concepts used by XVCL without the restrictions of XML language, which makes the understanding and coding easier.

4.3 Usage examples

Cpp+ is supported by a processor, the cpp+ processor. When we start the process we should determine one input file for processing. Usually we start the process of a framework with a so-called specification (SPC) file. In this file we determine the variables we will use during the process of cpp+ files (files instrumented with cpp+ command). The SPC specifies how to adapt the cpp+ files in order to accommodate required variants. After setting the variables we adapt the template file(s) in different way to get the preprocessed source code.

In this section we will see some simple usage examples of cpp+.

4.3.1 Adapting files

The cpp+ processor starts processing with the specification file. Cpp+ commands in the SPC, and in each subsequently adapted file, are processed in the sequence they appear. Whenever the processor encounters an **#adapt** command in the currently processed file, processing of the current file is suspended and the processor will start processing the adapted file. Once processing of the adapted file is completed, the processor resumes processing of the current file. In that way the processing ends when the cpp+ processor reaches the end of SPC file.

We can determine the output file of the processing with the **#output** <path> cpp+ command. Where <path> can be absolute or relative path. If we don't determine the output file, the output will be emitted to the default output, the input directories place as 'output'. It is

recommended to use the output command. In these simple examples we will use the default output.

Example: We have an SPC file, and we adapt A and C file. Furthermore the file A will adapt the file B. Cpp+ syntax:

```
SPC file:
  1:   before adapting A
  2:   #adapt A
  3:   after adapting A
  4:   before adapting C
  5:   #adapt C
  6:   after adapting C

file A:
  1:   content of A
  2:   before adapting B
  3:   #adapt B
  4:   after adapting B

file B:
  1:   content of B

file C:
  1:   content of C
```

The cpp+ processor starts at the first line of SPC file. If the processor finds a line without any command, transmits it to the output. In this way, the first line of the SPC file will appear in the output file. The second line tells the processor to start processing of the file named **A**. In the file **A** the first two lines will be transmitted to the output. In the third line of file **A** we can find another **#adapt** command referring to file **B**. Here starts the processing of file **B** which contains one line. This line will appear in the output, and we reached the end of file **B**. We continue from where the processing finished, from file **A** line: 4. The processing goes this way till we reach the end of the SPC file. We can see the order of processing the files on Figure 4.2.

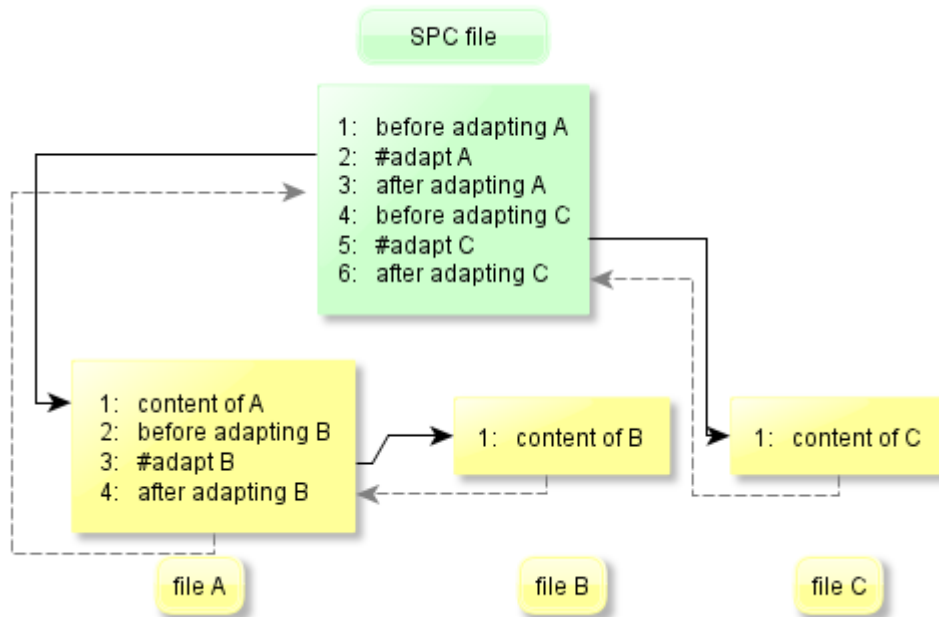


Figure 4.2: The process of the adapt example

The output of the above example will be:

output file:

```

1:   before adapting A
2:   content of A
3:   before adapting B
4:   content of B
5:   after adapting B
6:   after adapting A
7:   before adapting C
8:   content of C
9:   after adapting C

```

We can use the **#adapt** command in two forms in cpp+. First is what we used in the above example, with syntax `#adapt <file>`. Second form of the adapt command can be expanded with additional options. We can define some variables what we will need for processing the adapted file. We can define contents to *insert* in given *breakpoints* (as in 4.3.2 example), and we can use another commands what will be valid for the adapted file. The syntax of an extended adapt is:

```

#adapt: <file>
    additional information (#set, #while, #select, #insert
    commands)
#endadapt

```

It is important not to miss the semicolon “:” character after the **#adapt** keyword.

4.3.2 The break-insert mechanism

The body of extended **#adapt** command can contain **#insert** **#insert-before** and **#insert-after** commands. With these commands we can replace or extend the added *breakpoints* in the adapted file. The *breakpoints* are marked with the **#break** cpp+ commands.

The *breakpoints* (marked with **break** command) in an adapted file can be customized by **#insert**, **#insert-before** or **#insert-after** commands declared in the ancestor files (ancestor in adaption).

The insert commands modify adapted files at breakpoints identified by matching **#break** commands. This matching between **#insert** and **#break** commands is established by reference to the name of **#break** commands in the **#insert** command. We can use the **insert** commands as follows:

- The **#insert** command replaces the break-content inside its matching **#break** commands with its insert-content.
- The **#insert-before** command inserts its insert-content before the matching **#break** commands.
- The **#insert-after** command inserts its insert-content after the matching **#break** commands.

The **insert** commands are ended with an **#endinsert** command.

The next simple example will show how we can use the break-insert mechanism with cpp+:

```
SPC file:
1:      #adapt: A
2:          #insert-before breakX
3:              inserting before the breakpoint
4:          #endinsert
5:          #insert breakX
6:              inserting into the breakpoint
7:          #endinsert
8:          #insert-after breakX
9:              inserting after the breakpoint
10:         #endinsert
11:     #endadapt

file A:
1:      #break breakX
2:          default text
3:      #endbreak
```

The cpp+ processor starts the processing with the SPC file. The first line is an extended adapt command (**#adapt:**). Before starting the adapt of file **A** the processor executes the commands from the adapt body. In this case it will store 3 insertions.

- First: line 2 **#insert-before breakX**. This results in the next files adapted from here – directly or indirectly – the text from the **#insert-before** body will be placed before every occurrence of the breakpoint **breakX**.
- Second: line 5 **#insert breakX**. This results in the next files adapted from here – directly or indirectly – the text from breakpoint **breakX** will **replaced** with the text from the body of **#insert** command.
- Third: line 8 **#insert-after breakX**. This results in the next files adapted from here – directly or indirectly – the text from the **#insert-after** body will be placed after every occurrence of the breakpoint **breakX**.

The **#insert-before** and **#insert-after** commands do not replace the *default* text inside the breakpoint.

After the processor executed the commands in the adapt body – in this case stored the texts and given breakpoint to insert – starts the adapt of the file **A**.

The result file of the given example is the following:

```
result file:
1:   inserting before the breakpoint
2:   inserting into the breakpoint
3:   inserting after the breakpoint
```

If we delete the lines 5-6-7 from the SPC file – this means we don't use the '**#insert**' command – the result will be the following:

```
result file(2):
1:   inserting before the breakpoint
2:   default text
3:   inserting after the breakpoint
```

If **insert** commands refer to the same **break** by *break-name* from within multiple files or code fragments, only the **insert** command that is executed first in the processing flow will match this **break**. The remaining insert commands referring to the same break in other files or cpp+ code fragments will be ignored.

4.3.3 Cpp+ variables

In cpp+ we can declare variables with the **#set** command. The set command works similar to cpp's **#define** command. If the cpp+ processor finds a **#set** command it will store the variable's name and value. With the **#set** command we can declare *simple* and *multi-value* variables.

The *simple-value* can be an integer or a string (it is stored in the same way), for example:

```
#set X = 5
#set Y = John
#set text = "this is a text"
```

We must be mindful not to use a string variable in an arithmetic expression, it will lead to error. We can refer to the value of a variable with the '@' character. For example we refer to value of variable **X** with expression '@X'. We discuss the expressions later in this section.

With *multi-value* variables we can store multiple values with a variable name. We declare multi-value variables with the same command, listing the values separated by comma. For example:

```
#set X = 1,2,3
#set Y = one, two, three
```

4.3.4 Scoping rules

The scoping rules of variables in cpp+ are different than in C programming language. The first declaration of the variable will be the valid value, except if the redefinition is on the same scope level (file). The following examples will demonstrate this:

Example 1:

```
SPC file:
1:    #set X = 10
2:    #adapt A

file A:
1:    #set X = 20
2:    The value of X is: @X
3:    #adapt B

file B:
1:    #set X = 30
2:    The value of X is: @X
```

The Figure 4.3 illustrates the processing of this example:

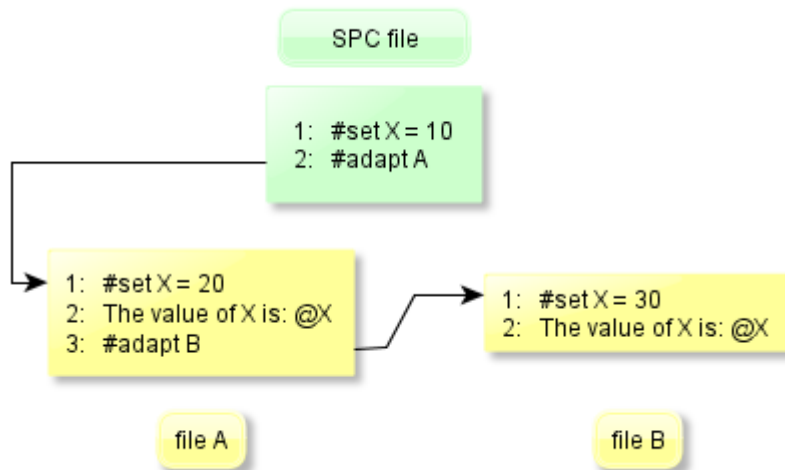


Figure 4.3: Processing of example 1

The output of the example will be the following:

Output file:

```

1: The value of X is: 10
2: The value of X is: 10

```

We can see, the **#set** commands in the adapted files were ignored because they are in a lower level. So the first declaration remains.

Example 2:

SPC file:

```

1: #set X = 10
2: #adapt A
3: #set X = 20
4: #adapt B

```

file A:

```

1: The value of X is: @X
2: #set X = 30
3: The value of X is: @X

```

file B:

```

1: The value of X is: @X
2: #set X = 30
3: The value of X is: @X

```

The Figure 4.4 illustrates the processing of this example:

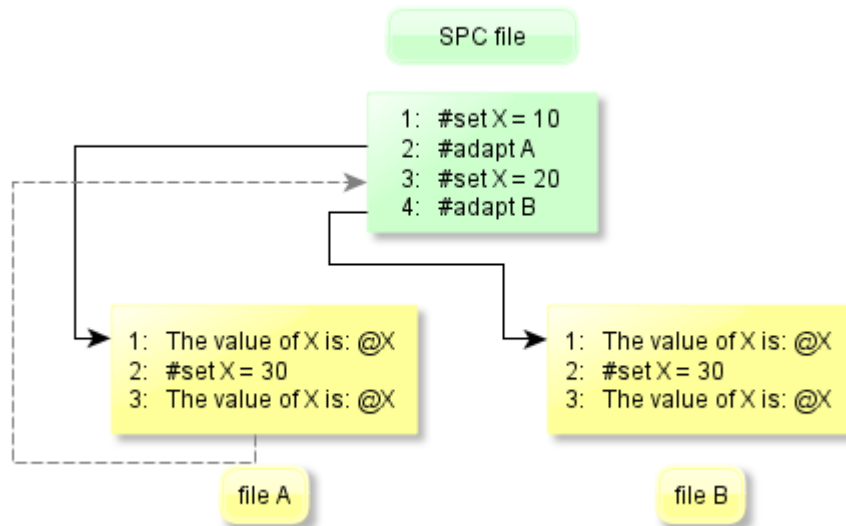


Figure 4.4: Processing of example 2

The output of the example will be the following:

Output file:

```

1:   The value of X is: 10
2:   The value of X is: 10
3:   The value of X is: 20
4:   The value of X is: 20

```

In this case the value of **X** has changed, because we had a redefinition in the same level where the file was declared, in the SPC file (line 3). The **#set** commands in file **A** and **B** were ignored.

4.3.5 The while and select commands

Cpp+ incorporates the XVCL's while and select commands. These commands are really important in variability management. With the while command we can implement a loop in the code, and with select we can separate individual code fragments depending on variables.

While:

The **#while** command iterates over its body in the following way: In each iteration uses the i^{th} value of the given multi-value variable. Starting with 1, **#while** implicitly increments the value of the index i by 1 in each iteration and terminates after processing the last value of variable. We can make a **#while** command with multiple variables by listing them after the command keyword separated by comma. In this case, all the multi-value variables listed here, must have the same number of values. The body of the **#while** command ends with an

#endwhile command. Values of the multi-value variables listed after the command keyword can be referenced in the body of the **#while** command. If used inside **#while** command, these multi-value variables behave like a single-value variables and can be referenced just like a single-value variable. This is illustrated in the following simple example.

```
1:  #set X = 1,2,3
2:  #set Y = a,b,c
3:  #while X, Y
4:    Value of X is @X and value of Y is @Y
5:  #endwhile
```

The output of this example will be the following:

```
1:    Value of X is 1 and value of Y is a
2:    Value of X is 2 and value of Y is b
3:    Value of X is 3 and value of Y is c
```

#while command is often used for code generation. For example, generating code that creates database tables, user interfaces buttons and menus, etc.

Dynamic invocation of multi-value variables is also possible in **#while** commands, as illustrated in example below:

```
1:  #set multi1 = A,B
2:  #set multi2 = C,D
3:  #set index = 1,2
4:  #while index
5:    #while multi?@index?
6:      #@multi@index?
7:    #endwhile
8:  #endwhile
```

The output of this example will be the following:

```
1: A
2: B
3: C
4: D
```

The expressions from line 4 and 5 will be explained in this section later.

Select:

With the **#select** command we can separate some code fragment depending on value of the selected variable. The syntax of the **#select** command is the following:

```
#select <variable>
      #option-undefined
```

```

        this will be executed if <variable> is not defined
#endoption-undefined
#option <value>
    this will be executed if value of <variable> is the given
    <value>
#endoption
#option <value2|value3>
    this will be executed if value of <variable> is <value2>
    OR <value3>
#endoption
...
#otherwise
    this will be executed if <variable> is defined, and none of
    the options corresponds to value of <variable>
#endotherwise
#endselect

```

In the **#select** body we can define zero or one **#option-undefined** clause, zero or more **#option** clauses and zero or one **#otherwise** clause.

The cpp+ processor checks **#option**-s in sequential order. If the value given in the option clause is the same as the variable's value, the **#option** command's body will be executed. We can use the pipe character “|” as OR in the given option. For example **<value2|value3>** will be evaluated as true if the value of the selected variable is ‘value2’ OR ‘value3’. The cpp+ processor executes all options clauses where the given value is the value of the selected variable.

We can use the **#select** and the **#while** command together to generate similar code fragments with some differences. In the body of while, we select with a select command for which value which code fragment we want to adapt.

A simple example: Suppose we want to generate similar classes with small differences. We adapt in multiple ways a generic class what defines the common part. In the SPC file we place a **#select** command inside the **#while** command, in the **#options** of **#select** command we specify the individual parts for the classes. Classes that require the same or no customizations will be generated in the **#otherwise** part of the select. In cpp+ code:

```

#set shape = Triangle, Circle, Rectangle, Square
#while shape
Class @shape {
    #select shape
        #option Triangle
            #adapt: genericShape
            customizations for Triangle class
            #endadapt
        #endoption
        #option Circle

```

```

        #adapt: genericShape
        customizations for Circle class
        #endadapt
    #endoption
    #otherwise
        #adapt: genericShape
        customizations for all remaining classes
        #endadapt
    #endotherwise
#endselect
}
#endwhile

```

4.3.6 Expressions

In cpp+ we can use and refer to the variables in some different ways. A direct reference to variable X is written @X. Each extra '@' symbol in the front of a variable name indicates a level of indirection. This means we can refer to the value of (value of X) with '@@X', and to the value of '@@X' with '@@@X'. As we can see in the next simple example:

```

1:    #set X = Y
2:    #set Y = Z
3:    #set Z = W
4:    @X
5:    @@X
6:    @@@X

```

The output of this example will be:

```

1:    Y
2:    Z
3:    W

```

The line 4 of the input is evaluated as value of X, the line 5 is evaluated as value of (value of X) => value of Y, and the line 6 is evaluated as value of (value of (value of X)) => value of (value of Y) => value of Z. This kind of reference to a variable is called *direct reference*.

XVCL introduces three types of expressions: the *name expression*, the *string expressions* and the *arithmetic expression*. We can use these kinds of expressions in cpp+ too.

A *name expression* can contain *variable references* (like ?@X?), and combinations of *variable references* (like ?@X@Y@Z?). We can see how the cpp+ processor evaluates an expression like this, in the next example. In cpp+ we write the name expressions between question marks '?'.

```

1:    #set X = Y
2:    #set Y = Z
3:    #set Z = W

```

```

4:      #set YW = U
5:      #set XU = Q
6:      ?@X@Y@Z?

```

The value of such a *name expression* (like in line 6 of the input) are computed from right to left evaluating the last '@' expression and concatenating with the other part of expression. In this case the evaluation will be the next:

- ?@X@Y@Z? : we replace '@Z' with its value 'W', and concatenate with the rest
- ?@X@YW? : we replace '@YW' with its value 'U', and concatenate with the rest
- ?@XU? : value of '@XU' = Q

A *string expression* can contain any number of *name expressions* intermixed with character strings. To evaluate a *string expression*, we evaluate the *name expressions* from the left to the right of the *string expression*, replace *name expressions* with their respective values and concatenate with character strings. For example a name expression can be "?@X@Y@Z?String?@XU?". We evaluate this expression in the following way:

- First we evaluate the first *name expression* ?@X@Y@Z?, this will be: Q.
- We concatenate the result with the other part of expression. Now the expression is: QString?@XU?.
- The next *name expression* is ?@XU? we replace this with its value: Q
- We concatenate the result with the other part of expression again. And the final result is: QStringQ

The cpp+ processor accept some valid basic *arithmetic expressions*. The expressions can contain '+', '-', '*', '/' operators and nested parenthesis can be used. The arithmetic expression must yield to integer, or the cpp+ processor will report an error. (For example if we have (3 + 'a'), and a in not defined.)

The cpp+ processor accepts arithmetic expressions in lines which are intended to be in output and in **#set** commands. As we can see in the next examples:

```

#set X = 3
#set Y = 5
@X + @Y
#set Z = (?@X? + ?@Y?) - 2
#set W = ?@X?, ?@X?+1, (?@X?+1)+1

```

We implemented the cpp's conditional expansions in cpp+. In this way we can refer to variables defined in cpp+ within **#if** - **#elif** - **#else** - **#endif** constructions. The syntax is the same as in cpp, we can use the above-mentioned *arithmetic expressions* and *boolean expressions*. For boolean expressions we can use the '&&', '||', '<', '>', '<=', '>=' and the 'defined' operators regarding to variables defined in cpp+. The syntax of the cpp+'s conditional expansions are:

```
#if <expression>
    code
#elif <expression>
    code
#elif <expression>
    code
#else
    Code
#endif
```

The **#if** body can contain zero or more **#elif** commands with expression, and zero or one **#else** command.

Where the <expression> can contain:

- Integer constants
- Character constants that represent an integer e.g., "123"
- Arithmetic operators for addition, subtraction, multiplication, division, and logical operations (&& and ||).
- References to cpp+ variables and expressions. cpp+ variables should contain an integer value and cpp+ expressions should be arithmetic expressions that yield integer. cpp+ variables and expressions are expanded before actual computation of the expression's value begins.
- Uses of the `defined` operator, which lets us check whether cpp+ variables are defined (1) or not (0).

An example using the cpp+'s **#if** command:

```
#set X = 3
#set Y = 5
#set Z = 12
#if defined X && Z <= (2 * 6) || Y > 40
    first case occurred
    #adapt A
#elif Y + X < 2
    second case occurred
```

```
        #adapt B
    #else
        #adapt C
    #endif
```

We also implemented the cpp's **#ifdef** and **#ifndef** commands for the same reason as the **#if** - **#elif** - **#else** commands.

4.4 Evaluation

We could see in this section how simple is the cpp+ syntax, and how easy is to understand its functionality. The cpp supplemented with cpp+'s features can be a better and more effective way to implement the variability management of a software system. In addition usage of cpp+ makes the system much easier to understand and expand.

There are some cpp commands which are not implemented yet in cpp+, and there are some which we do not plan to implement. We planning to pass the **#include** command to the C preprocessor to execute it. Replacement of **#include** commands would not make the code easier to understand, and cpp can also include library files which are not textual contexts.

We did not implement yet the concept of *function-like macros* in cpp+. So these **#define** commands will also be passed to the C preprocessor.

It can be some situations, when we would like to keep some cpp commands in the output files of cpp+ processing. For example if we would like to make a portable code, and leave in the code some conditional directives.

If we use the **#\<cpp command>** syntax in cpp+, the given command will be in the output of cpp+ processing as cpp command, so the cpp+ processor just delete the **'\'** escape character. In this way we can have in the output some commands, what we want to pass to cpp processor.

Cpp+ is also effective in improving program maintenance. It is a well-known fact that redundant code obstructs program understanding and maintenance. With cpp+ we can get rid of these redundancies. In the next section we will see a proof of this.

5 Linux Kernel case study

5.1 Introduction to Linux

Linus Torvalds – the developer of Linux – studied at the University of Helsinki. Linus did use Minix, a low-cost Unix created as a teaching aid, but he was discouraged by the inability to easily make and distribute changes to the system’s source code (because of Minix’s license) and by design decisions made by Minix’s author. In response to his predicament, Linus decided to write his own operating system [22]. He developed the first version of Linux in 1991 as an operating system for computers powered by the Intel 80386 microprocessor, which at the time was a new and advanced processor.

Linus began by writing a simple terminal emulator, which he used to connect to larger Unix systems at his school. His terminal emulator evolved and improved. In a short time, Linus had a full-fledged Unix on his hands. He posted an early release to the Internet in late 1991. Use of Linux took off, with early Linux distributions quickly gaining many users. More important to its initial success, however, is that Linux quickly attracted many developers, hackers adding, changing, improving code. Because of the terms of its license, Linux swiftly evolved into a collaborative project developed by many.

Today, Linux is a full-fledged operating system also running on Alpha, ARM, PowerPC, SPARC, x86-64 and many other architectures. It runs on systems as small as a watch to machines as large as room-filling super-computer clusters.

Linux is a Unix-like system, but it is not Unix. That is, although Linux borrows many ideas from Unix and implements the Unix API (as defined by POSIX and the Single Unix Specification), it is not a direct descendant of the Unix source code like other Unix systems. Where desired, it has deviated from the path taken by other implementations, but it has not forsaken the general design goals of Unix or broken standardized application interfaces.

Linus remains the creator of Linux and the *maintainer* of the kernel, progress continues through a loose-knit group of developers. The Linux kernel is licensed under the GNU General Public License (GPL) version 2.0. Consequently, you are free to download the source code and make any modifications.

The basics of a Linux system are the kernel, C library, toolchain, and basic system utilities, such as a login process and shell. A Linux system can also include a modern X Window System implementation including a full-featured desktop environment, such as GNOME. Thousands of free and commercial applications exist for Linux.

5.2 Linux kernel and cpp

5.2.1 Kernel (of an operating system) in general

The kernel of an operating system is the core internals, the software that provides basic services for all other parts of the system, manages hardware, and distributes system resources. The kernel is sometimes referred to as the *supervisor*, *core*, or *internals* of the operating system.

Typical components of a kernel are interrupt handlers to service interrupt requests, a scheduler to share processor time among multiple processes, a memory management system to manage process address spaces, and system services such as networking and interprocess communication

The user applications execute in *user-space*. They see a subset of the machine's available resources and can perform certain system functions, directly access hardware, access memory outside of that allotted them by the kernel, or otherwise misbehave. When executing kernel code, the system is in kernel-space executing in kernel mode. When running a regular process, the system is in user-space executing in user mode. Applications running on the system communicate with the kernel via *system calls* (see Figure 5.5.1).

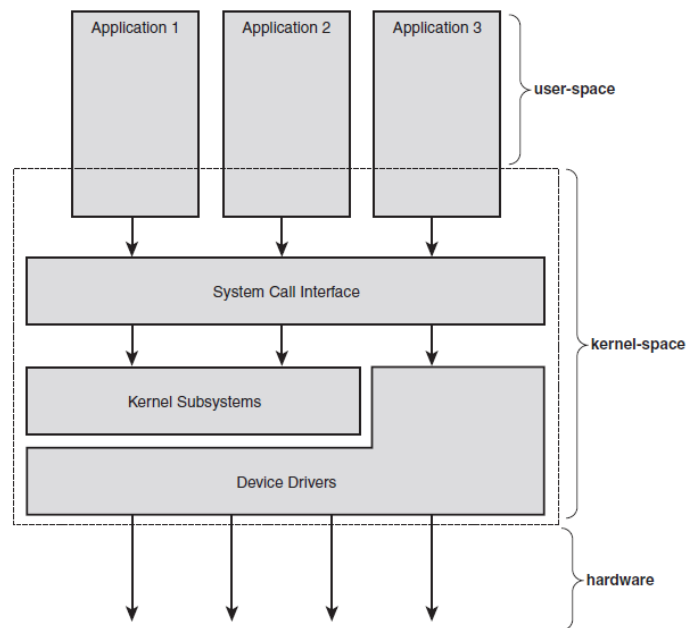


Figure 5.5.1 Relationship between applications, the kernel, and hardware. [30]

The kernel also manages the system's hardware. Nearly all architectures, including all systems that Linux supports, provide the concept of *interrupts*. When hardware wants to communicate with the system, it issues an interrupt that literally interrupts the processor, which in turn interrupts the kernel. A number identifies interrupts and the kernel uses this number to execute a specific interrupt handler to process and respond to the interrupt. In many operating systems, including Linux, the interrupt handlers do not run in a process context. Instead, they run in a special interrupt context that is not associated with any process.

We can divide kernels into two main design forms: the *monolithic kernel* and the *microkernel* [23][24][25].

The **Monolithic kernels** are the simpler design of the two, and all kernels were designed in this manner until the 1980s. Monolithic kernels are implemented entirely as a single process running in a single address space. Consequently, such kernels typically exist on disk as single static binaries. All kernel services exist and execute in the large kernel address space. Communication within the kernel is trivial because everything runs in kernel mode in the same address space: The kernel can invoke functions directly, as a user-space application might. Proponents of this model cite the simplicity and performance of the monolithic approach. Most Unix systems are monolithic in design.

Microkernels, on the other hand, are not implemented as a single large process. Instead, the functionality of the kernel is broken down into separate processes, usually called *servers*. All the servers, though, are separated into different address spaces. Therefore, direct function invocation as in monolithic kernels is not possible. Instead, microkernels communicate via *message passing*: An interprocess communication (IPC) mechanism is built into the system, and the various servers communicate with and invoke “services” from each other by sending messages over the IPC mechanism. The Windows NT kernel (on which Windows XP, Vista, and 7 are based) and Mach (on which part of Mac OS X is based) are examples of microkernels.

5.2.2 Linux kernel

Linux is a monolithic kernel. The Linux kernel executes in a single address space entirely in kernel mode. Linux also borrows much of the good from microkernels: Linux boasts a modular design, the capability to preempt itself (called *kernel preemption*), support for kernel threads, and the capability to dynamically load separate binaries (kernel modules) into the kernel image. Conversely, Linux has none of the performance-sapping features that curse microkernel design: Everything runs in kernel mode, with direct function invocation – not message passing – the modus of communication. Nonetheless, Linux is modular, threaded, and the kernel itself is schedulable.

Because Linux is not based on any specific Unix variant, Linus and its company can pick and choose the best solution to any given problem, or at times, invent new solutions.

We always can find two states of the Linux kernel: stable and development. Stable kernels are production level releases suitable for widespread deployment. New stable kernel versions are released typically only to provide bug fixes or new drivers. Development kernels, on the other hand, undergo rapid change where (almost) anything goes. As developers experiment with new solutions, the kernel code base changes in often drastic ways.

5.2.3 How cpp is used in Linux kernel

Linux kernel’s current extension approach uses conditional preprocessor directives to create configuration-dependent code. With this approach we can easily include or exclude parts of the source code.

Configuring the kernel is a required step before building it. Because the kernel offers myriad features and supports a varied basket of hardware, there is a *lot* to configure. We can do this via configuration options. The configuration options are used both to decide which files to build and to manipulate code via preprocessor directives.

Configuration of the kernel is controlled by configuration options built into the kernel's configuration system. Each configuration option is represented by a variable prefixed with "CONFIG_" as in CONFIG_FOO. From one version of Linux to another, many of the changes made to the kernel code are in the form of new configuration options and the addition or modification of configuration dependent code. When the kernel is built, the values of configuration options dictate what files are compiled and which code within those files is included. Configuration options manipulate the code base through preprocessor directives. The preprocessor directives include **#define**, **#undef**, **#include**, **#pragma**, **#line**, and the six conditional directives: **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif**. The conditional directives let blocks of code be included or excluded from the code at compile time. The Linux coding style provides guidelines on the use of the preprocessor directives. We can see below two code examples from the kernel code.

kernel/arch/arm/mm/fault.c:

```
515:    /* FSR definition */
516:    #ifdef CONFIG_ARM_LPAE
517:    #include "fsr-3level.c"
518:    #else
519:    #include "fsr-2level.c"
520:    #endif
```

kernel/drivers/Bluetooth/hci_ldisc.c:

```
572:    static void __exit hci_uart_exit(void)
573:    {
574:        int err;
575:
576:        #ifdef CONFIG_BT_HCIUART_H4
577:            h4_deinit();
578:        #endif
579:        #ifdef CONFIG_BT_HCIUART_BCSP
580:            bcsp_deinit();
581:        #endif
582:        #ifdef CONFIG_BT_HCIUART_LL
583:            ll_deinit();
584:        #endif
585:        #ifdef CONFIG_BT_HCIUART_ATH3K
586:            ath_deinit();
587:        #endif
588:
```

```

589:          /* Release tty registration of line discipline */
590:          if ((err = tty_unregister_ldisc(N_HCI)))
591:              BT_ERR("Can't unregister HCI line discipline (%d)",
err);
592:      }

```

5.3 Clones in Linux kernel

Large number of variation points at which we must embed code related to various configuration options in the base code is the main source of difficulties of configuring options with cpp. The only way to counter this problem is to reduce the size of the base code.

Suppose that we find some Linux source files, each more than 200 LOC, with much similarity among those files. Suppose further that we can extract similarities to form a generic, parameterize template for those files. By instantiating template parameters in different ways, we can derive any of the found files. The number of variation points is bound to be smaller in a template than in the original files in which similar code is replicated. Also understanding of how various configuration options affect the base code is likely to be easier from the template perspective rather than by looking at the original source files.

Cpp+ offer a solution along the above lines.

We searched for software clones in Linux Kernel 3.3.6 with Clone Miner [27] to see if we can find groups of similar files that would enable us to simplify managing configuration options.

Clone Miner is written in C++, and it has its own token-based simple clone detector. With Clone Miner we can find more types of structural clones in program code of the given system. The biggest similarities in the Linux kernel can be found in the *arch* and *drivers* subdirectories.

arch: The *arch* subdirectory contains all of the architecture specific kernel code. It has further subdirectories, one per supported architecture, for example *i386* and *alpha*.

drivers: All of the system's device drivers live in this directory. They are further sub-divided into classes of device driver, for example *block*.

In the *arch* subdirectory the similarities appeared between some similarly implemented architecture specific code fragments, for example in the memory management of the *sparc* and *x86* architecture.

In the drivers subdirectory the similarities appeared in the drivers of similar hardware, for example in the code of some network card drivers. We decided to continue the search in the *drivers* subdirectory. Here the biggest similarity between files can be found in the *drivers/input/touchscreen* folder.

Nine files share the same structure, consisting big similarities between each other. The files are:

- *dynapro.c*
- *fujitsu_ts.c*
- *gunze.c*
- *hampshire.c*
- *inexio.c*
- *mtouch.c*
- *penmount.c*
- *touchit213.c*
- *touchright.c*
- *touchwin.c*

The files in average consist of more than 200 lines of code. The mentioned source files are drivers of different touchscreen tools. They are responsible for describing how the operating system should communicate with the given connected hardware (in this case touchscreen). Due to the fact these code fragments are intended to serve the same purpose, the files have a lot of similarities in the code and in the structure of the code too. The files differ in the way how the function is implemented which is responsible for connecting and initialize the data of the driver structure. Some drivers have additional data and of course they differ in the driver's name, author, etc.

For example in all mentioned files we can find the next code fragment in the procedure which is responsible for disconnecting the device:

```
gunze.c:
96:  static void gunze_disconnect(struct serio *serio)
97:  {
98:      struct gunze *gunze = serio_get_drvdata(serio);
99:
100:     input_get_device(gunze->dev);
101:     input_unregister_device(gunze->dev);
102:     serio_close(serio);
103:     serio_set_drvdata(serio, NULL);
104:     input_put_device(gunze->dev);
105:     kfree(gunze);
106: }
```

This fragment of code appears in all mentioned files in the same way. The difference is just in the words highlighted (in bold) which referring to the name of the device. We can see this on the next picture (Figure 5.2) captured from a diff tool.

<pre> 81 82 /* 83 * tr disconnect() is the opposite of tr_connect() 84 */ 85 86 static void tr_disconnect(struct serio *serio) 87 { 88 struct tr *tr = serio_get_drvdata(serio); 89 90 input_get_device(tr->dev); 91 input_unregister_device(tr->dev); 92 serio_close(serio); 93 serio_set_drvdata(serio, NULL); 94 input_put_device(tr->dev); 95 kfree(tr); 96 } 97 </pre>	<pre> → ← </pre>	<pre> 91 92 /* 93 * gunze disconnect() is the opposite of gunze_connect() 94 */ 95 96 static void gunze_disconnect(struct serio *serio) 97 { 98 struct gunze *gunze = serio_get_drvdata(serio); 99 100 input_get_device(gunze->dev); 101 input_unregister_device(gunze->dev); 102 serio_close(serio); 103 serio_set_drvdata(serio, NULL); 104 input_put_device(gunze->dev); 105 kfree(gunze); 106 } 107 </pre>
--	------------------	---

Figure 5.2 Similarities between touchright and gunze's disconnect procedure

The files have more similar code fragments. The differences are mostly in some keywords. We can achieve a simpler representation (which is easier to understand and to expand) of the listed files, with non-redundant, generic representation.

5.4 Sketch of solution with cpp+

The nine files discussed in the previous section share common structure which can be defined as cpp+ template. Any differences among files become template parameters (so-called variation points) marked with cpp+ directives. The actual differences among files are defined in SPC which also specifies the process of generating 9 files from the template (Figure 5.3).

In the above case we can make a generic representation easily. First we make a file where we define and store the differences in cpp+ variables. This will be the specification file. This file will contain mostly **#set** commands, which are intended to define variables.

After we made the specification file, we can use it to adapt the template file (the non-redundant generic representation) with given customization, which will represent the desired files. For example if we want the result to be the header of the above mentioned *disconnect* procedure for 'tr', and 'gunze' variables, we can make the following steps.

SPC file:

```

1: #set var = tr, gunze
2: #while var
4:     #output ?@var?.c
3:     #adapt commonFile
4: #endwhile

```

```
template:
1: static void ?@var?_disconnect(struct serio *serio)
```

In the specification file we define and set the value of the variable '*var*'. The variable will be a multi-value variable with values '*tr*', and '*gunze*'.

With a **#while** command we can iterate over a multi-value variable, the n^{th} iteration of the **#while** command will use the n^{th} value of the given variable(s). In this example the body of **#while** command will be processed two times, first with the value '*tr*', after that with the value '*gunze*'. We can find the explanation of this command in section 4.3.1.

With the **#output** command we can define the path and the name of the output. In this example it will be the value of variable *var* with '.c' extension. In this case we will have a *tr.c* and a *gunze.c* file.

With the **#adapt** command we can process other cpp+ files. We can find a detailed explanation of this command and examples in section 4.3.1.

The result of this example will be the next:

```
tr.c:
1: static void tr_disconnect(struct serio *serio)
gunze.c:
1: static void gunze_disconnect(struct serio *serio)
```

With the above mentioned approach we can make one SPC file for the 9 above-listed files, and the template will be the body of the files where the differences will be replaced with variables and breakpoints. On Figure 5.3 we can see the sketch of the approach with cpp+.

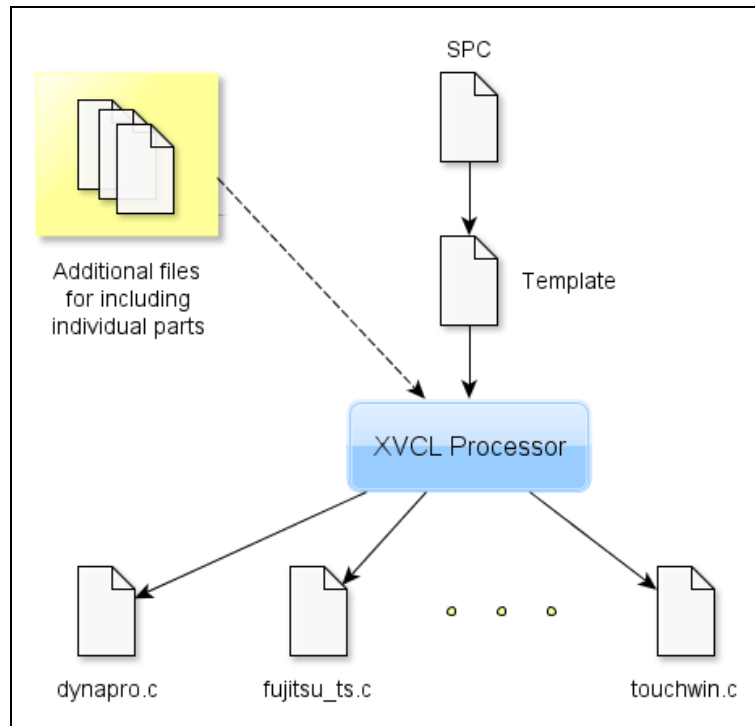


Figure 5.3 Production of the 9 files with cpp+

Even though the files have really big similarities, some parts differ in more percentage than it worth to replace with a common parametric code fragment. We resolve this problem with CPP+'s break-insert mechanism. The specification file will decide for every loop, in which breakpoints which inserts will be executed. The inserts contains an adapt command referring to a file which contains the insertable content. The insertable contents can be in separate files, or in one file separated by a select command, depending on style of inserting. This makes the example easier to understand. With this approach we represent one of the original file with the content of more files (including the specification and common file) during the processing flow. One example for this approach:

Specification file:

```

1:  #set driverName = dynapro, mtouch
2:  #while driverName
3:      #output output/?@driverName?.c
4:      #adapt: drivers/common
5:          #select driverName
6:              #option mtouch
7:                  #insert processData
8:                  #adapt
drivers/?@driverName?/processData
9:                  #endinsert
10:             #endoption
11:          #endselect
12:      #endadapt
13:  #endwhile

```

In the first line of example we set the variable `'driverName'` to multi-value with two values `'dynapro'` and `'mtouch'`.

In line 2, we iterate over the values of `'driverName'`, so the lines 3-12 will be executed two times, in the first iteration the expression `?@driverName?` will be evaluated as `'dynapro'` in the second will be `'mtouch'`.

In line 3 we set the output like in the previous example. Line 4 is similar to the previous example too, but here the `#adapt` command is an extended adapt command. In this part we will set the inserts what we need for the common file's breakpoints.

In the lines 5-11 there is a `#select` command, what helps in selection of in which loop which insert we want to use. In this example in first loop we jump over the `#select` command, because it doesn't have option for `'dynapro'`. In second iteration the lines 6-10 will be executed. So we insert in the breakpoint `'processData'` the content of `'drivers/mtouch/processData'` file.

If we want another file where we will replace the same code fragment, we just have to extend the option command's condition. For example we want the same construct for variable `'third'` we have to extend the line 1 to:

```
1:      #set driverName = dynapro, mtouch, third
```

and the line 6 to:

```
6:      #option mtouch|third
```

The code fragment in option will be executed if value of `?@driverName?` is `'mtouch'` OR `'third'`.

Of course we must have the file `'drivers/third/processData'`.

With these two small modifications we achieved to have a third file between the outputs `'third.c'` implementing the common part of the drivers with a modification in the `'processData'` breakpoint. This proves extending fragments made with cpp+ is really easy, and the structure is easy to understand.

5.5 Evaluation

We made the cpp+ implementation of the above mentioned files from Linux. In this solution we represent the mentioned 9 files with cpp+ code and generate the c code with the implemented cpp+ processor.

The original representation of the driver files in total consists of 1860 lines of code. The representation with cpp+ consists of 732 lines of code. With cpp+ we could eliminate 61% of the code using the commonalities between files and parameterization.

In addition, cpp+ template shows the essential structure shared by 9 files. This is useful for program comprehension. It also makes it easier to trace the impact of any modification.

The cpp+ representation make the code much easier to understand. We can look at individual parts separately. If we need to make a similar driver, like in the above example, we can choose which parts we need in it and which parts not. In this representation if we need a new driver's source file, we just need to add to the cpp+ code some words, and add the individual code of the non-common parts. If we don't use cpp+ we need to write the whole source code of driver from scratch.

6 Conclusion

As I mentioned earlier, we can consider cpp+ as an extension of cpp. We can use the commands and features of cpp, and some additional features too. We can define multi-value variables, and implement loops in the preprocessing of the code. Cpp+ provides the insert-break mechanism to include or exclude individual parts of the code. For conditional compilation the language contains the if and the select mechanism.

The power of cpp+ is mostly in its simplicity and scalability. It is easy to understand, to learn how to use and easy to expand the existing codes. First we need effort to make the representation of a project with cpp+, but after that makes the maintenance and expansion of the code much easier than working with the pure source code.

We are planning to improve the language with some useful additional features, to make the usage easier and more effective. We are planning to make more case studies, and to use cpp+

on more projects. More experience would help a lot to find the right way, how we can improve the language.

References

- [1] Julio Sincero and Wolfgang Schröder-Preikschat. "The Linux Kernel Configurator as a Feature Modeling Tool" Department of Computer Science 4 Friedrich-Alexander University Erlangen-Nuremberg
- [2] Danilo Beuche and Mark Dalgarno. "Software Product Line Engineering with Feature Modelling"
- [3] Myllymäki, "Variability Management in Software Product Lines" 3.12.2001
- [4] A. Karhinen, A. Ran, and T. Tallgren, "Configuring designs for reuse," Proc. Int. Conf. on Soft. Eng., ICSE'97, Boston, MA., 1997, pp. 701-710
- [5] H. Spencer, and G. Collyer, "ifdef Considered Harmful, or Portability Experience with C News," USENIX, Summer 1992 Technical Conference, San Antonio, Texas, June, 1992, pp. 185-197.
- [6] Bassett, P. Framing software reuse - lessons from real world, Yourdon Press, Prentice Hall, 1997
- [7] XVCL website <http://xvcl.comp.nus.edu.sg/cms/>
- [8] Wong, T.W., Jarzabek, S., Myat Swe, S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Proc. ACM Symposium on Software Reusability, SSR'01*, Toronto, Canada, May 2001, pp. 164-172
- [9] Stan Jarzabek, Paul Bassett, Hongyu Zhang and Weishan Zhang "XVCL: XML-based Variant Configuration Language"
- [10] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2003, Helsinki, pp. 237-246
- [11] Stan Jarzabek, Hamid Abdul Basit and Damith C. Rajapakse "Beyond Templates: a Study of Clones in the STL and Some General Implications," ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA.
- [12] Case Study: Computer Aided Dispatch (CAD) System Product Line, <http://xvcl.comp.nus.edu.sg/xvcl/cad/CAD.html>
- [13] Richard M. Stallman, Zachary Weinberg, "The C Preprocessor," For gcc version 4.8.0 (pre-release) 2011
- [14] C Preprocessor Wikipedia page: http://en.wikipedia.org/wiki/C_preprocessor
- [15] Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann, "Partial Preprocessing C Code for Variability Analysis," Philipps University Marburg 2011
- [16] Jörg Liebig, Sven Apel, Christian Kästner and Michael Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines" 2010
- [17] P. Clements, and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002
- [18] C. Kästner, S. Apel, and D. Batory, "A Case Study Implementing Features Using AspectJ," Proc. Int. Soft. Product Line Conf., SPLC'07, Kyoto, Sept. 2007, pp.223-232
- [19] P. Clements, and D. Muthig, (Editors) Proc. Workshop on Variability Management – Working with Variation mechanisms, at the 10th Software Product Line Conference (SPLC-2006), Fraunhofer IESE-Report No 152.06/E Version 1.0, Germany, October 15, 2006
- [20] source: <http://snow.nl/dist/htmlc/ch01s03.html>, Compiling a Linux Kernel
- [21] Unix Wikipedia page: <http://en.wikipedia.org/wiki/Unix>
- [22] Linux Wikipedia page: <http://en.wikipedia.org/wiki/Linux>
- [23] Benjamin Roch and TU Wien: "Monolithic kernel vs. Microkernel"
- [24] Monolithic kernel Wikipedia page: http://en.wikipedia.org/wiki/Monolithic_kernel
- [25] Microkernel Wikipedia page: <http://en.wikipedia.org/wiki/Microkernel>
- [26] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski and Krzysztof Czarnecki, "The Variability Model of The Linux Kernel"
- [27] Stan Jarzabek and Hamid Abdul Basit, "A Data Mining Approach for Detecting Higher-level Clones in Software," 24.02.2007

- [28] C. Brabrand and M. I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 31–40. 2002.
- [29] Y. Padioleau. Parsing C/C++ Code without Pre-Processing. In Proc. Int’l Conf. Compiler Construction (CC), pages 109–125. 2009.
- [30] Robert Love, “Linux Kernel Development, Third Edition,” June 2010
- [31] Batory, D., Sarvela, J.N. and Rauschmayer, A. 2003 “Scaling Step-Wise Refinement,” Proc. Int. Conf. on Soft. Eng., ICSE’03, May 2003, Portland, Oregon, pp. 187-197
- [32] Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. “Aspect-Oriented Programming,” Europ. Conf. on Object-Oriented Prog., Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242
- [33] Jarzabek, S. and Li, S. “Eliminating redundancies with a “Composition with Adaptation” meta-programming technique,” Proc. Europ. Soft. Eng. Conf. and ACM SIGSOFT Symp. on Foundations of Soft. Eng. ESEC-FSE’03, Helsinki, pp. 237-246 Sept. 2003
- [34] Pettersson, U., and Jarzabek, S. “Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach,” ESEC-FSE’05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September 2005, Lisbon, pp. 326-335
- [35] Jarzabek, S. Xue, Y., Zhang, H. and Lee, Y. "Avoiding Some Common Preprocessing Pitfalls with Feature Queries," Proc. 16th Asia-Pacific Software Engineering Conference, APSEC’09, Penang, Dec. 2009, pp.283-290
- [36] Kästner, C., Apel, S. and Kuhlemann, M.: Granularity in Software Product Lines. In ICSE’08, 2008
- [37] J. Yang and S. Jarzabek “Applying a Generative Technique for Enhanced Reuse on J2EE Platform,” 4th Int. Conf. on Generative Programming and Component Engineering, GPCE’05, Sep 29 - Oct 1, 2005, pp. 237-255.