XCcpp: Preprocessing without Some Common Pitfalls

Stan Jarzabek¹, Hongyu Zhang², Youpeng Lee², Yinxing Xue¹, Naveed Shaikh¹

¹School of Computing
National University of Singapore
Singapore 117543
stan@comp.nus.edu.sg

²School of Software
Tsinghua University
Beijing 100084, China
hongyu@tsinghua.edu.cn

Abstract

Preprocessing (e.g., cpp) is a simple technique to handle program variants by including/excluding variant feature code to/from a base program. When variant features grow in number, they affect the base code and each other in many places, and feature interactions lead to hidden dependencies among many code segments. Code becomes heavily instrumented with preprocessing directives, difficult to understand, test, maintain and reuse. In the paper, we describe XCcpp, a system that to certain extent alleviates some of these problems, by extending preprocessing mechanisms, and providing a query-based visual environment for analysis of feature interaction patterns. XCcpp is a subset of XVCL generative technique and tools [27]. We use a sizable study to illustrate problems that such extended preprocessing can and cannot solve. XCcpp concepts can enhance any preprocessing system known to us, such as cpp or m4, providing similar benefits, and added values as a variation mechanism in reuse via Product Line approach. We compare XCcpp with compositional and annotative approaches proposed in research.

1. Introduction

The many ways in which preprocessing has been used have been discussed in [11]. Here, we are interested in applying preprocessing to manage program variants by including/excluding variant feature code to/from a base program. In this context, program variants form a software Product Line (SPL) [7], and preprocessing becomes a mechanism to organize variant features around the base code, and selectively include features into custom software products.

Industrial practice revealed serious problems with this kind of reuse [16][24]. As the number of variant features grows, programs instrumented with macros become difficult to understand, test, maintain and reuse. It is difficult to trace feature-related code, and to understand or change program in general. Managing features with #ifdefs is technically feasible, but is error-prone and does not scale [16]. Karhinen et al [16] observe that feature management at the implementation level is only bound to be complex. Authors describe problems from Nokia projects in which preprocessing and file-level configuration management were used to manage features. They propose to use design as means to counter problems. Similar problems with preprocessing were also reported in a research project FAME-DBMS [18][22].

Today’s mainstream approach to industrial Product Line reuse is motivated by the above experiences. Much emphasis is put on design-level means - component architectures - to manage variant features [5][7]. During product derivation, we reuse common design (PL architecture) shared by SPL members.

One of the technical challenges in SPL approach is to support variation of features [7][8]. It is common to use additional variation mechanisms such as preprocessing, configuration files or wizards, in addition to component/architecture design, to manage variant features at the level of component implementation. Mappings between variant features, reusable components and specific variation points in components affected by a given feature selection may be complex. This complexity is inherent in the nature of the reuse problem. Managing features that require many fine-grained changes at arbitrary places in many reusable components remains a challenge. Problems magnify in the presence of feature dependencies, when the presence or absence of one feature affects the way other features are to be implemented [21]. Despite many benefits of architecture- and component-based approaches to reuse, managing features that have fine-granular impact on many reusable components requires extensive manual, error-prone customizations during product derivation [10].

Researchers proposed Feature-Oriented Programming (FOP) [21] and experimented with
Aspect-Oriented Programming (AOP) [19] to realize feature management by modularizing features, and then composing them into the base code [24][1]. A recent study has revealed difficulties in using AspectJ as a FOP realization technique [17]. Another study [18] hinted at difficulties to compose fine-granular-impact features with AHEAD [4].

Old problems that once preprocessing could not solve, still show their face in modern design and component technologies applied in industrial practice, and hinders penetration of approaches proposed in research into software reuse practice. The above experiences suggest that in cases of fine-granular-impact features, it may be difficult to avoid keeping feature code together with reusable components (or base code).

Along those lines of thinking, in [18], authors relaxed FOP’s requirement for feature modularization, and revisited the idea of keeping feature-related code together with the base code. A tool called CIDE provides visual means for understanding and manipulating features. CIDE annotates feature code on the Abstract Syntax Tree (AST) of the base, and shows code related to various features using different colors.

In this paper, we show a solution that achieves similar benefits by simpler means that do not require parsing. For that, we extend conventional preprocessing into a system that we call XCcpp. By “conventional preprocessing”, we mean mechanisms such as conditional compilation and macros that are found in popular systems (e.g., cpp or m4). We describe XCcpp notation and capabilities independent of any particular preprocessing system, using a subset of XVCL generative meta-programming system and its supporting tools implemented in the XVCL Workbench (XML-based Variant Configuration Language [27][13]).

XCcpp promotes preprocessing from add-in macro notation/mechanism that merely instruments source code and manipulates locally at the place of macro invocation, to a first-class meta-level notation with its own AST internal representation that can be automatically processed and analyzed.

XCcpp keeps feature code together with the base code in a meta-level program representation, and uses conventional preprocessing-like directives to include/exclude variant features into/from the base code. Each variation point in the base code (i.e., point affected by some features) is marked with a specific combination of features that affect that point. All variation points that are associated with a certain feature are inter-linked by means of meta-variables with global scope. These global controls allow XCcpp to realize and automate feature selection and analysis.

XCcpp meta-level program representation is processed before compilation to derive custom products.

In addition, XCcpp includes tools that traverse, analyze and visualize the meta-program to help developers understand, reuse and maintain features and the base code. XCcpp includes a query-based system that allows developers to specify which features and in what context they wish to analyze. A visualization tool highlights feature code in the analysis context specified by queries.

Like most of preprocessors, XCcpp is language-independent. The concepts of XCcpp can be implemented on top of any preprocessing system, and applied to any programming language.

In the paper, we use DB Berkeley, a system used in earlier studies on feature management with AspectJ [17] and CIDE [18], to illustrate preprocessing problems (Section 3) and our approach to alleviating them (Section 4). By using the same case study, we can easier grasp similarities and differences among various approaches to feature management. We evaluate our method in Section 5.

We position contributions of this work as follows:

- While pitfalls of preprocessing have been widely discussed, the evidence is mostly anecdotal. We conducted qualitative and quantitative analysis of preprocessing problems in a sizeable, real system, showing the sources, nature and scale of the problems. We also pointed to problems that we believe are inherent in approaches that attempt to manage features in the base code. The problems that we studied and the results apply to all the preprocessing systems that we know of.
- We proposed simple extensions to preprocessing to alleviate some common pitfalls. These extensions can be easily implemented on top of existing preprocessing systems, providing benefits similar to those of XCcpp. We believe presented solutions can allow preprocessing to serve better as a variation mechanism [8] complementing component/architectural approaches to Product Line development [5][7].

2. Basic preprocessing solution in XCcpp

XCcpp notation is a subset of XVCL, but it has the essential characteristics of preprocessing systems such as cpp or m4.

The base code and features are organized into meta-components called x-frames. The overall solution shown in Figure 1 is called an x-framework. The top-most SPECification x-frame (SPC) defines a required feature selection, and x-frames at the bottom contain base classes with feature code embedded under <option>s of <select> commands.
X-frames are processed by **XVCL Processor** to derive a program variant - an SPL member - that implements a selection of features specified in **SPC**. Processing starts at **SPC**, and continues within and across x-frames in depth-first order via `<adapt>` links. During this traversal of x-frames, XVCL Processor interprets XVCL commands and emits code contained in x-frames to the output files. `<adapt>` is analogous to cpp’s `#include`. However unlike `#include`, `<adapt>` can specify customizations to be applied to an adapted x-frame. The same x-frame can be adapted in different ways at different adaptation points. While we do not make use of this adaptive reuse capability of XVCL in the solution presented here, it is quite essential in building more advanced reusable software representations.

We mark each variation point in the base code (i.e., point affected by features) with `<select>` command that indicates how various combinations of features affect that point. We link all the variation points that have to do with a certain feature by means of meta-variables. Meta-variables have global scope, and their values propagate down the x-frames along `<adapt>` links. These global controls also allow us to realize and automate feature manipulation.

**Figure 1. Feature management with XVCL: a sample x-framework**

Suppose feature A interacts with class `Base-X`. X-frame `Base_X.xvcl` contains `<select>` command with `<option A>` that marks the variation point. This is a simple case of a feature affecting the base code without interactions with other features. Such cases are also nicely handled by preprocessing `#ifdef`.

X-frame `Base_Y.xvcl` shows the case of a feature interaction pattern. The relevant variation point contains a number of `<option>`s marked by the names of features that interact at that point. For example, `<option A+C+D>` contains code for features A, C and D interacting at this point.

The above solution is put to work by means of meta-variables and meta-expressions. We create meta-variables for each feature. Their values are `<set>` in **SPC**. In our case, we select features A and D, so meta-variable `a` is `<set>` to “A” and `d` is `<set>` to “D”. Meta-variables for unwanted features are `<set>` to null string “ ”.

**Values of meta-variables propagate down to the `<adapt>`ed x-frames, and are used to identify `<option>`s for relevant feature interactions. The pattern of feature interaction at various variation points is identified by meta-variable `v` that controls `<select>`. Notation `@v` means a reference to variable `v`. In x-frame `Base_Y.xvcl` feature interaction involves A, C and D. As we selected features A and D, meta-variable `v` will be `<set>` to “A” in `Base_X.xvcl`, and to “A+D” in `Base_Y.xvcl`, where the value of `v` is defined as concatenation (operator “+”) of values of meta-variables `a`, `c`, and `d`.

In the above solution, `<select>` is analogous to cpp’s `#if.. #elif .. #else .. #endif`. The same net effect can be achieved with `#ifdef`, however the representation may involve deeply nested `#ifdef` that multiply alternative code paths [24]. Some of these paths are meaningless, but they still should be understood and tested (the latter being an unachievable goal). Flat nesting structure of `<select>` avoids these extra problems. Command `<set>`
is analogous to #define except that values propagate globally across adapted source files contained in x-frames. XCpp expressions may not have direct counterpart in some preprocessing systems, but this is a minor extension.

As not all the combinations of features can be legally selected for any given custom product, it is a good practice to validate a given feature selection before the customization process starts. We developed suitable validation techniques using formal methods Z, Alloy and OWL DL [25, 26].

3. Basic preprocessing solution to DB Berkeley and its problems

We start by analyzing a basic preprocessing solution to manage features in the Berkeley DB. Berkeley DB is an open source database engine (http://www.oracle.com/technology/products/berkeley-db). Many features can be optionally included into custom DB systems using runtime mechanisms. In that sense, Berkeley DB forms a Product Line, whose members implement different selections of features. In earlier studies, Berkeley DB was converted into a Product Line in which features were managed at the design-time (e.g., before compilation) with AspectJ [17] and CID [18]. In our study, we also managed features at the design-time with XCpp. By using the same study, we have a good opportunity to compare various approaches.

3.1. Overview of our study

The Berkeley DB consists of five subsystems: access methods to create and access the database, B-tree to store data as key/value pairs, caching and buffering to increase database performance, concurrency and transaction to handle concurrent and rollback facility, and finally persistence layer.

38 features such as IO, LookAheadCache or DiskFullHandler can be optionally included into custom DB systems.

In our study, we addressed 22 Berkeley DB features ranging from simple to complex. First, we analyzed the semantics of all features that we wanted to work with, and how they affected the base code.
We wrapped every base class affected by features into an x-frame. An excerpt of the x-framework for Berkeley DB is shown in Figure 2. XVCL variable features_selected in SPC contains features to be included in a DB variant, in case of our example, features IO, LookAheadCache and DiskFullHandler. SPC adapts an x-frame BaseManager.xvcl, which in turns adapts all x-frames for DB base classes. For clarity, in Figure 2, we have shown only two of such x-frames, namely FileProcessor.xvcl and LogBuffer.xvcl. These x-frames contain base classes affected by selected features as defined in features_selected. In addition, BaseManager.xvcl also adapts feature-specific x-frames. This is because for some of the features, besides adding small fragments of code at variation points in base classes upon feature selection, we must also add whole class(es) to the DB base. Class Evictor in Figure 2 exemplifies this situation, and so does feature STATISTICS that adds classes StatsConfig and BtreeStats to the DB base (not shown in Figure 2).

3.2. Patterns of feature impact on DB base

Here, we analyze various forms of feature impact, and discuss how they complicate preprocessing solution.

Type 0. Preprocessing-friendly features

Features that affect small number of base classes at small number of variation points, and without interacting with other features are easily handled by preprocessing. Such features may affect the base code by adding member variables and methods to certain base classes. Each such feature impact is easily handled by placing these member variables and methods within <select-option> block in affected base classes. Figure 3 shows the case of feature DiskFullHandler affecting the base class LogBuffer in that way.

Some features require adding new classes to the DB base, such as already mentioned features EVICTOR and STATISTICS. Yet other features require adding a new parent to a base class. Features with that kind of impact on the base code, no matter of their granularity, are easily handled with preprocessing.

Among 22 features in DB that we addressed in the experiment, 5 features were of Type 0 and did not create major problems for preprocessing solution.

Type I. Many variation points in a base class

First type of complication occurs when the number of variation points at which features affect a given base class grows. The impact may come from one or more features.

```
public class FileProcessor {
    // ...
    private boolean processFile(...) {
        // ...
        LookAheadCache lookAheadCache = new LookAheadCache();
        // ...
        LookAheadCache.add(getFileOffset(...));
        // ...
        if (lookAheadCache.isFull()) {
            processLN(..., lookAheadCache, ...);
        }
        // ...
    }
    // ...
    private processLN(..., LookAheadCache lookAheadCache, ...) {
        int offset = lookAheadCache.getOffSet();
        // ...
    }
```

Figure 4. Impact of LookAheadCache on base class methods

Figure 4 shows the impact (underlined text) of feature LookAheadCache on class FileProcessor. This feature requires changes to method signatures to pass context information from one method to another. The method processLN takes LookAheadCache object as a parameter and is used to pass context information from processFile. This creates a semantic connection between the methods processLN and processLN. If the feature LookAheadCache is not selected and there is still LookAheadCache-independent context information that has to be passed to processLN from processFile, the connection between the two methods should remain intact.

Figure 5 shows how preprocessing handles the impact of feature LookAheadCache on base class FileProcessor. We defined two signatures of the method processLN for the cases when LookAheadCache is selected and when it is not. The code also required refactoring so that if LookAheadCache is not selected, the context information can still be passed from one method to another.
Class FileProcessor contains 23 variation points, 6 of which mark the impact of feature LookAheadCache.

Among other examples, 12 features affect class EnvironmentImpl inducing 38 variation points, and 10 feature affect class FileManager, inducing 40 variation points. The number of variation points per class ranges from 1 to 35, with average 5.72.

For feature impact of Type I, the base code becomes densely populated with `<select>`s, which is one of the often mentioned drawbacks of using preprocessing to handle program variants.

For feature impact of Type II, the base code becomes scattered over many classes. For example, feature LookAheadCache contributes complexity to the preprocessing solution despite the fact that it affected only two base classes at the total of 10 variation points.

Another dimension of complexity appears when the impact of one feature becomes scattered over many variation points, and spreads through many base classes. For example, feature MemoryBudget affects 32 DB base classes at a total of 183 variation points.

Other features of Type II include Statistics (34 variation points), CheckSum and Evictor (each with 27 variation points), and CriticalEviction (23 variation points).

For Type II, feature code becomes scattered across many variation points in base classes. Even if feature impact at each variation point is small, to understand the feature, we must find all the relevant variation points, and analyze feature code in multiple contexts. When modifying a feature, we must propagate changes to all the relevant variation points. Maintenance becomes difficult and error-prone.

**Type III. Feature dependencies and interactions**

A feature \( f_1 \) depends on feature \( f_2 \) if \( f_1 \) refers to code of \( f_2 \), or if the presence or absence of \( f_2 \) in a program variant affects implementation of \( f_1 \).

Feature dependencies cause feature interactions. In preprocessing solution, feature interactions show as variation points with `<option>`s labeled with names of interacting features, or - in more complex interaction situations - as multiple `<option>`s under a `<select>` command.

For example, feature EvictorDaemon depends on Evictor. In Figure 6, features Evictor and EvictorDaemon affect the base class EnvironmentImpl. They interact in the method requestShutdownDaemons, at the variation point `<option> Evictor+EvictorDaemon</option>`.

For example, feature CriticalEviction interacts with feature Evictor, and feature MemoryBudget interacts with feature CriticalEviction. The net result of those interactions shows at variation points in class Evictor. If we select feature CriticalEviction method `doCriticalEviction` must be included into the class Evictor. If at the same time we select feature MemoryBudget, method `doCriticalEviction` must be modified with code related to that feature.
Feature code scattering (Type II) and feature interactions lead to hidden dependencies among many variation points. These dependencies have to be understood to modify code without unwarranted side-effects. Examples in Figure 7 and Figure 8 show a hidden dependency induced by feature interactions. Features CriticalEviction and MemoryBudget affect class Evictor, which is only included if the Evictor feature is selected. Therefore, once the variation point that includes the class Evictor is removed, all variation points introduced by the features CriticalEviction and MemoryBudget in class Evictor will be automatically affected.

```java
public class Evictor {
    ...
    <set v= @CriticalEviction+@MemoryBudget />
    <select v> 
    <option CriticalEviction>
    public void doCriticalEviction() 
    throws DatabaseException {
        doEvict(SOURCE_CRITICAL, true);
    }
    </option>
    <option CriticalEviction+MemoryBudget> 
    public void doCriticalEviction() 
    throws DatabaseException {
        MemoryBudget mb = envImpl.getMemoryBudget();
        long currentUsage = mb.getCacheMemoryUsage();
        long maxMem = mb.getCacheBudget();
        ...
    }
    </select>
    ...
}
```

**Figure 7. Feature interactions (2)**

In Berkeley DB, we find 38 instances of feature interactions, scattered across 7 base classes. Type III features multiply problems of feature understanding and maintenance beyond what we observed for Type II features. Feature-interaction related code appears in contexts that are even more difficult to analyze.

### 3.3. Summary of preprocessing problems

Features of Types I, II and III result in base code densely populated with preprocessing directives. Feature code becomes spread over many variation points. To understand these features, we must find and analyze feature code variants in multiple contexts. Feature interactions further complicate the problem as interactions often lead to hidden dependencies among many variation points which have to be understood to correctly modify code.

In the context of feature management in a base program, preprocessing typically leads to the following problems:

**Problem 1.** Difficult to understand and maintain feature and base code. Difficult to trace the impact of changes.

**Problem 2.** Difficult to add new features. Difficult to see how new feature affects the base code and interacts with other features.

**Problem 3.** Difficult to reuse existing features in development of new program variants. Problems of extracting feature code are amplified by the need to handle variation in feature requirements: Suppose that two program variants P1 and P2 have slightly different requirements for the same feature. We need extra control parameters to handle this, which increases complexity of the preprocessor-based representation of a software Product Line.

### 4. XCcpp: feature queries and visualization

During maintenance, e.g., when we want to enhance a certain feature, we need understand how a given feature affects the whole system. In particular, we need to find all the feature code segments spread across many base classes, other features that a given feature interacts with, and the relevant variation points.

XCcpp offers queries and visualization to help developers find feature code segments and analyze them in the many contexts where they occur. As XCcpp is a subset of XVCL [27][13], we use XVCL Workbench tools for that. We formulate queries in FQL (Frame Query Language) and show query results in Workbench’s IDE (Figure 10).

In FQL, we can ask queries such as “which base classes are affected by feature f? and at which variation points?”, “which features interact with f?”, “in which base classes and at which variation points feature f1 interacts with feature f2?”. FQL is an SQL-like notation. We write queries in terms of XCcpp elements such as x-frames, `<select>`, `<option>` or metavariables. Queries can refer to relationships among these elements that are meaningful in the context of feature analysis. Queries follow the format shown in Figure 8. Keywords are in bold.

**Figure 8. Format of feature query**

Query ::= declare (<XCcpp-element> x;)*
        select <query-results>
        where <query-conditions>

Clause **declare** introduces names of XCcpp elements which are used in the rest of the query. Clause **select** lists query results, i.e., XCcpp elements we want to find. Clause **where** defines query conditions that constrain the result in terms of their participation in relationships and attribute values of XCcpp elements.
For example, query shown in Figure 9 finds all the code of feature MemoryBudget, and all classes affected by this feature. The two query conditions say that we are interested in all the x-frames ‘x’ that contain \(<\text{option}>\) labeled with feature name MemoryBudget. Since all the variation points at which MemoryBudget affects classes appear as \(<\text{option}>\)s marked with relevant feature names, this query effectively finds the desired result.

\[
\begin{align*}
\text{declare} & \ x\text{-frame } x; \ option \ o; \\
\text{select} & \ x, \ o \\
\text{where} & \ o.f\text{-names} = \text{"MemoryBudget"} \\
\text{and} & \ \text{Contains} \ (x,o)
\end{align*}
\]

**Figure 9. Finding feature code in base classes**

The result of the query is fed into Visualizer tool of XVCL Workbench which shows x-frames affected by feature MemoryBudget. By selecting various x-frames, a developer can now examine all the relevant variation points.

In Figure 10, the right-bottom window indicates that total 183 variation points were found in 32 base class x-frames. A developer selects an x-frame from the list, and can examine (in the large right-top window) the base class in which code related to MemoryBudget has been highlighted.

Interactions in various x-frames can then be examined using XVCL Workbench, as before.

\[
\begin{align*}
\text{declare} & \ option \ o \\
\text{select} & \ x.name, \ o \\
\text{where} & \ o.f\text{-names}=\text{"CriticalEviction\text{*MemoryBudget\text{*}"} \\
\text{and} & \ \text{Contains} \ (x,o)
\end{align*}
\]

**Figure 11. Finding feature interactions (1)**

The two queries of Figure 12 find all the variation points where Evictor affects base classes or interacts with other features.

\[
\begin{align*}
\text{declare} & \ x\text{-frame } x; \ option \ o; \\
\text{select} & \ x, \ o \\
\text{where} & \ o.f\text{-names} = \text{"Evictor"} \\
\text{and} & \ \text{Contains} \ (x,o)
\end{align*}
\]

**Figure 12. Finding feature interactions (2)**

This query returns 28 variation points located in 12 x-frames.

Any pattern of feature interactions can be located in that way and feature code can be analyzed in multiple contexts of their occurrence. We can find all feature interaction instances specifying “*+*” as a search condition.

**Figure 13. X-frame tree for query evaluation**

To analyze XCcpp meta-program and to evaluate queries, we build an XCcpp Abstract Tree (AST) for each x-frame. XCcpp-AST represents all the metaprogram information needed for feature analysis: Nodes in XCcpp-AST correspond to commands such as \(<\text{select}>\), \(<\text{option}>\), \(<\text{adapt}>\) or \(<\text{set}>\). Attributes of command nodes contain variable names, their values and feature names at respective \(<\text{option}>\)s. Other XCcpp-AST nodes contain segments of base and feature
that becomes heavily instrumented with preprocessing directives (features of Type I, Section 3.2): Queries and visualization can selectively reveal certain features, while hiding others that are not relevant to a given analysis context. XCpp can locate feature code, selectively show features at their variation points where they affect the base code or interact with other features. Visualization helps developers understand the scale of feature impact, and analyze feature code in multiple contexts at which they occur. This helps one locate scattered features (Type II), and analyze feature interactions (Type III).

Suppose we want to enhance feature MemoryBudget. For that, we may need to switch among 183 variation points where MemoryBudget affects 32 base classes. Furthermore, as MemoryBudget interacts with 6 features including Evictor and CriticalEviction. A change to MemoryBudget may have impact on these features. We may need to further examine Evictor code at 28 variation points in 12 classes and CriticalEviction code at 23 variation points and 8 classes. Feature modification requires rapid switches analysis contexts. Feature queries and visualization help in that.

A limitation of our DB Berkeley study with XCpp is that we have not evaluated the actual impact of XCpp on maintenance/reuse effort. Given that we do not have full understanding of DB design decisions, we felt that we were not in the position to conduct a sound experiment to facilitate such evaluation.

While XCpp can ease feature understanding and maintenance, the essential difficulty of having to analyze multiple feature contexts at the many variation points remain. It still hinders feature maintenance and adding new features, as all these contexts must be understood before implementing any change. Modifying features for specific program variants without affecting other program variants introduces more variation points and more control parameters that further complicate understanding, reusing and maintenance of program variants in XCpp representation.

Despite these limitations, we trust XCpp improves usability of preprocessors, increasing their value as variation mechanisms complementing reuse in software Product Lines.

We believe that in general, radical improvements of approaches that attempt to manage features in the base code are difficult to achieve. As Karhinen et al. [16] observed, working at the base code level constrains us from applying design-level means to manage features, and feature management at the implementation level is bound to be complex.

Other than architectural design used in SPL approaches to reuse [5][7], we used generic design as means to manage features. XVCL provides simple yet effective mechanisms for unrestrictive forms of parameterization, capable of representing any software repetition patterns - similar program structures of any kind and granularity - as generic, adaptable, reusable meta-components (x-frames). These generic meta-components are designed so that features become their parameters. XVCL Processor instantiates generic meta-components with feature parameters to generate concrete program components that implemented required selected features.

This form of feature management and reuse has been introduced in Frame Technology™ [3]. XVCL, based on frame concepts, has been applied in lab studies [2][12][28] and industrial projects [20][30]. The approach is comprehensibly covered in [13]. Unlike XCpp, full-fledged XVCL is more difficult to adopt in standard industrial software development.

6. Related Work

The many ways of using preprocessors and their common pitfalls have been widely discussed [11][16][24]. In this paper, we illustrated preprocessing problems in a study on feature management in Berkeley DB, and summarized problems reported by others.

Research community proposed Feature-Oriented Programming (FOP) [21] as an approach to feature management for reuse. FOP is based on feature modularization, and a mechanism for feature composition into a base program. One of the motivations of FOP is to support software Product Lines. Mixin technique [23] has been widely used for FOP, with AHEAD [4] being its most advanced realization. AHEAD provides powerful solution for feature management in many situations, but may not be geared for fine-granular impact features [18]. A
number of authors also proposed Aspect-Oriented Programming (AOP) [19] to realize feature management in compositional way [24][1]. Using AOP, features are modularized as aspects (advices and introductions) and then weaved (feature composition) into a base program. A recent study has revealed difficulties in using AspectJ as a FOP realization technique [17].

In view of the above findings, Kastner et al. [18] relaxed the requirement for feature modularization, and revisited the idea of keeping feature-related code together with the base code. A tool called CIDE (Colored IDE) provides visual means for understanding and manipulating features. It helps programmers to work with features, providing functions for examining, injecting and excluding feature-related code to/from the base program. CIDE represents a base program as an Abstract Syntax Tree (AST), which makes it language-dependent. Feature-related code annotates AST nodes. CIDE shows feature-related code using different colors. Unlike preprocessing approaches, it does not obfuscate the source code. However, some feature impacts cannot be handled by CIDE. An example is situation in Figure 5 where we need to provide alternate implementation of a method depending on feature selection.

In contrast to CIDE, XCpp uses programming language-independent preprocessing-like program representation to achieve similar goals. XCpp represents preprocessing constructs as well as base/feature code segments as an AST. By promoting preprocessing to the first-class meta-programming notation and mechanism, offering tools that help in localizing and visualizing feature-related code, and show abstract views focusing on certain features under analysis, while hiding other features. We analyzed preprocessing problems and illustrated XCpp benefits in a study of a sizable Berkeley DB system.

The strengths of XCpp are simplicity of means to manage features, language-independence, and reliance of well-known preprocessing mechanisms. Our solution can be easily adapted to enhance capabilities of existing preprocessing systems. Despite advancements in component/architecture support for reuse, preprocessing still plays a role as a complementary variation management technique [5][7]. We believe XCpp contributes new values in this context.

In evaluation, we stress that XCpp avoids only accidental complexities of preprocessing (in Brook’s sense [6]). Essential complexities do not go away, in particular, scattering of feature code and hidden dependencies among features create challenges for feature maintenance and reuse. In addition, problems of evolving base code and features, independently of changes done to custom products derived from the base, are difficult to handle.

In the paper, we pointed to the general nature of the difficulties of addressing the problems in the frame of approaches that rely on managing feature code in a base program. In future work, we plan to integrate XCpp feature management solution described in this paper with our earlier reuse experiences with XVCL. We plan to improve our solution by integrating XCpp meta-program analysis with analysis of the code at base/feature nodes.

7. Conclusions
Preprocessors are still used, despite their well-known pitfalls. We described simple extensions that make preprocessing more useful. Poor readability, scattering of feature code across the base program, and feature interactions are often mentioned among the reasons why preprocessing leads to code that is difficult to understand, test, maintain and reuse. We presented XCpp system that helps one locate and visualize scattered feature code and feature interactions, alleviating some of the above preprocessing pitfalls. XCpp promotes preprocessing to the first-class meta-programming notation and mechanism, offering tools that help in localizing and visualizing feature-related code. We achieve this with simple extensions to commonly used preprocessing systems, and tools that understand meta-level program representation. XCpp applies query and visualization techniques to navigate programmers through feature-related code, and show abstract views focusing on certain features under analysis, while hiding other features. We analyzed preprocessing problems and illustrated XCpp benefits in a study of a sizable Berkeley DB system.

The strengths of XCpp are simplicity of means to manage features, language-independence, and reliance of well-known preprocessing mechanisms. Our solution can be easily adapted to enhance capabilities of existing preprocessing systems. Despite advancements in component/architecture support for reuse, preprocessing still plays a role as a complementary variation management technique [5][7]. We believe XCpp contributes new values in this context.

In evaluation, we stress that XCpp avoids only accidental complexities of preprocessing (in Brook’s sense [6]). Essential complexities do not go away, in particular, scattering of feature code and hidden dependencies among features create challenges for feature maintenance and reuse. In addition, problems of evolving base code and features, independently of changes done to custom products derived from the base, are difficult to handle.

In the paper, we pointed to the general nature of the difficulties of addressing the problems in the frame of approaches that rely on managing feature code in a base program. In future work, we plan to integrate XCpp feature management solution described in this paper with our earlier reuse experiences with XVCL. We plan to improve our solution by integrating XCpp meta-program analysis with analysis of the code at base/feature nodes.
References


