Proc. 16th Asia-Pacific Software Engineering Conference, APSEC'09, Penang, Dec. 2009, pp.283-290

Avoiding Some Common Preprocessing Pitfalls with Feature Queries

Stan Jarzabek and Yinxing Xue
Department of Computer Science
National University of Singapore
Singapore
stan@comp.nus.edu.sg

Hongyu Zhang and Youpeng Lee
School of Software
Tsinghua University
China
hongyu@tsinghua.edu.cn

Abstract - **Preprocessors (e.g., cpp) provide simple means to manage software product variants by including/excluding required feature code to/from base program. Feature-related customizations occur at variation points in base program marked with preprocessing directives. Problems emerge when the number of inter-dependent features grows, and each feature maps to many variation points in many base program components. Component-based and architecture-centric techniques promoted by a Software Product Line approach to reuse help us contain the impact of some features in small number of base components. Still, accommodating other features into product variants requires fine granular code changes in many components, at many variation points. Fine granular code level changes are often handled by preprocessors, which becomes a source of well-known complications during component customization for reuse. In this paper, we show how some of the common preprocessing problems can be alleviated with a query-based environment that assists programmers in analysis of features handled with preprocessor's directives. We describe problems of preprocessing that can be aided by tool like ours, and problems that we believe are inherent in approaches that attempt to manage features in the base code.**

Keywords – preprocessing, product variants, reuse, variation mechanisms

I. Introduction

The many ways in which preprocessors, e.g., cpp or m4, are used have been discussed in [1]. Here, we are interested in applying preprocessing to manage product variants by including/excluding variant feature code to/from a base program. In this context, product variants form a Software Product Line (SPL) [2], and preprocessing becomes one of the reuse mechanisms for program customization.

Industrial practice revealed serious problems with this kind of reuse [3][4]. 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 [3]. In [3], Karhinen et al. observed that feature management at the implementation level only is bound to be complex. They described problems from Nokia projects in which preprocessing and file-level configuration management were used to manage features. They proposed to use design as means to overcome these problems. Similar problems with preprocessing were also reported in a research project FAME-DBMS [5][6].

Today's mainstream approach to reuse is motivated by the above experiences. Much emphasis is placed on design-level means - component architectures - to manage product variants in reuse-based way [7][2]. Still, mappings between features, reusable components and specific variation points in components affected by features are often 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 implemented [8].

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 [9]. It is common to use variation mechanisms such as preprocessing, configuration files or wizards, in addition to component/architecture design, to manage features at the level of the component code.

In this paper, we describe XCpp - a preprocessing system extended with query-based feature analysis - that mitigates some of the problems in feature management with preprocessors. Feature queries allow developers to specify which features and in what context they wish to analyze. XCpp finds program files affected by features of interest, and relevant variation points in those files. We discuss preprocessing problems XCpp can solve, and problems it cannot solve.

A solution described in this paper can be implemented on top of any preprocessing system, and applied to any programming language.

To present the concepts of our solution, we emulate preprocessing in XVCL (XML-based Variant Configuration Language) [10]. XVCL is a variation mechanism to build adaptable, reusable meta-components and to organize them into Product Line architecture for reuse [11]. XVCL commands to handle variability at the detailed level of component code are similar to preprocessing systems such as cpp or m4. We use only this subset of XVCL commands in this paper.

We illustrate our solution with Berkeley DB, a sizable system, used in earlier studies on feature management with AspectJ [12] and CIDE [5]. By using the same study, we have an extra opportunity to compare various approaches to feature management.

Contributions of this work are as follows:

- While pitfalls of preprocessing have been widely discussed, the evidence is mostly anecdotal. In our study, we conducted analysis of preprocessing problems in a sizeable, real system, showing the 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 propose 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 XCpp. We believe our solutions can allow preprocessing to serve better as a variation mechanism [13] complementing component/architectural approaches to Product Line development [7][2].

In Section II, we describe our notation to model product variants managed by a preprocessor. In Section III, we discuss common preprocessing problems and illustrate them with examples from Berkeley DB. XCpp is described in Section IV. We evaluate merits and limitations of XCpp in Section V.

II. Preprocessing as a Variation Mechanism

It is useful to categorize features that differentiate product variants as follows: *Fine-grained features* affect many base components, at many variation points; *coarse-grained features* can be contained in base components that are included into a custom product that needs such features; *mixed-grained features* involve both fine- and coarse-grained impact.

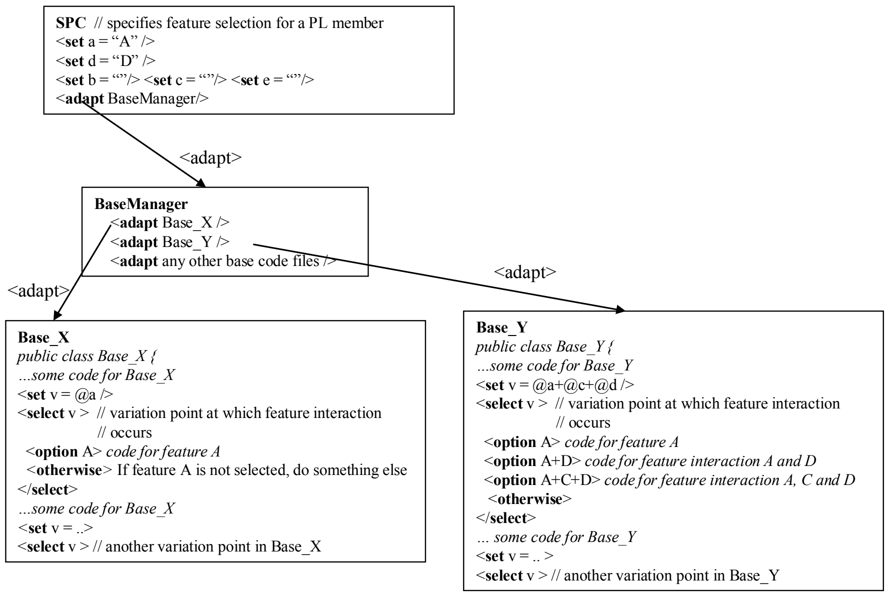
Coarse-grained features can be easily accommodated into product variants with **#include** directives placed in base components, or using a build tool such as *make*. Fine-grained feature code is kept together with the base code under conditional compilation directives (e.g., #ifdef). Each variation point in the base code (i.e., point affected by some features) corresponds – in more or less explicit way - to a specific combination of features that affect that point.

In our model preprocessing notation, <**select-option**> commands mark variation points. Suppose features A, B, C and D can be optionally included into product variants, members of some software Product Line. Suppose further that **Base_X** and **Base_Y** are two reusable base components (source files) for that Product Line. Feature A interacts with **Base_X** as shown in Figure 1. **Base_X** contains <**select**> command with <**option** A> that marks that variation point. This is a simple case of a feature affecting the base code without interactions with other features. Such cases are also easily handled by preprocessing directives **#ifdef**.

The above solution is put to work by means of parameters and expressions. Each feature is represented by a parameter. For example, parameter *a* represents feature A, *b* – feature B, and so on. The top-most SPeCification file, **SPC,** <**set**>s values of parameters and in that way specifies which features we need in a product variant. In Figure 1, we wish to select features A and D, so parameter *a* is <**set**> to "A" and *d* is <**set**> to "D". Parameters for unwanted features are <**set**> to null string " ". Notation **@v** means a reference to variable *v*.

To derive a product variant that implements selected features, base components are processed starting with **SPC**, in depth-first order via <**adapt**> links. During this traversal, the Processor interprets commands and emits code contained in base components to the output files, just as any preprocessor does. <**adapt**> is analogous to cpp's **#include**.

During processing, values of parameters propagate down to the <**adapt**>ed base components, and are used to identify <**option**>s relevant to selected features. Features that affect a base component at a given variation point are identified by parameter *v* that controls <**select**>. In SPC of Figure 1, we selected features A and D, so in **Base_X**, parameter *v* is <**set**> to "A", and code under <**option** A> is included into the product variant.



```
SPC  // specifies feature selection for a PL member
<set a = "A" />
<set d = "D" />
<set b = ""/> <set c = ""/> <set e = ""/>
<adapt BaseManager/>
```

```
BaseManager
   <adapt Base_X />
   <adapt Base_Y />
   <adapt any other base code files />
```

```
Base_X
public class Base_X {
...some code for Base_X
<set v = @a />
<select v >  // variation point at which feature interaction
                 // occurs
  <option A> code for feature A
  <otherwise> If feature A is not selected, do something else
</select>
...some code for Base_X
<set v = ..>
<select v > // another variation point in Base_X
```

```
Base_Y
public class Base_Y {
...some code for Base_Y
<set v = @a+@c+@d />
<select v >  // variation point at which feature interaction
                 // occurs
  <option A> code for feature A
  <option A+D> code for feature interaction A and D
  <option A+C+D> code for feature interaction A, C and D
   <otherwise>
</select>
... some code for Base_Y
<set v = .. >
<select v > // another variation point in Base_Y
```

Figure 1. Managing fine-grained features in base components with preprocessor

Base_Y includes a variation point with interacting features A, C and D. As we selected features A and D, parameter *v* is <**set**> to "A+D" in **Base_Y**, where the value of *v* is defined as concatenation (operator '+') of values of parameters *a*, *c*, and *d*.

Effectively, each variation point is clearly marked with names of features that affect that point, and interact at that point. All the variation points that are associated with a certain feature are inter-linked by means of parameters with global scope.

Our <**select**> is analogous to the **#if** .. **#elif** .. **#else** .. **#endif** directive. It is functionally equivalent to **#ifdef**, but avoids problems of nested **#if**'s analyzed in [4]. <**set**> command is analogous to the #define directive except that values propagate globally across base components. In cpp, parameters set at the command line also propagate to all the preprocessed files. Expressions at <**option**>s may not have direct counterpart in some preprocessing systems, but this is a minor extension, and can be easily implemented to any preprocessor.

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. Such validation can be done using formal methods Z, Alloy and OWL DL [14][15].

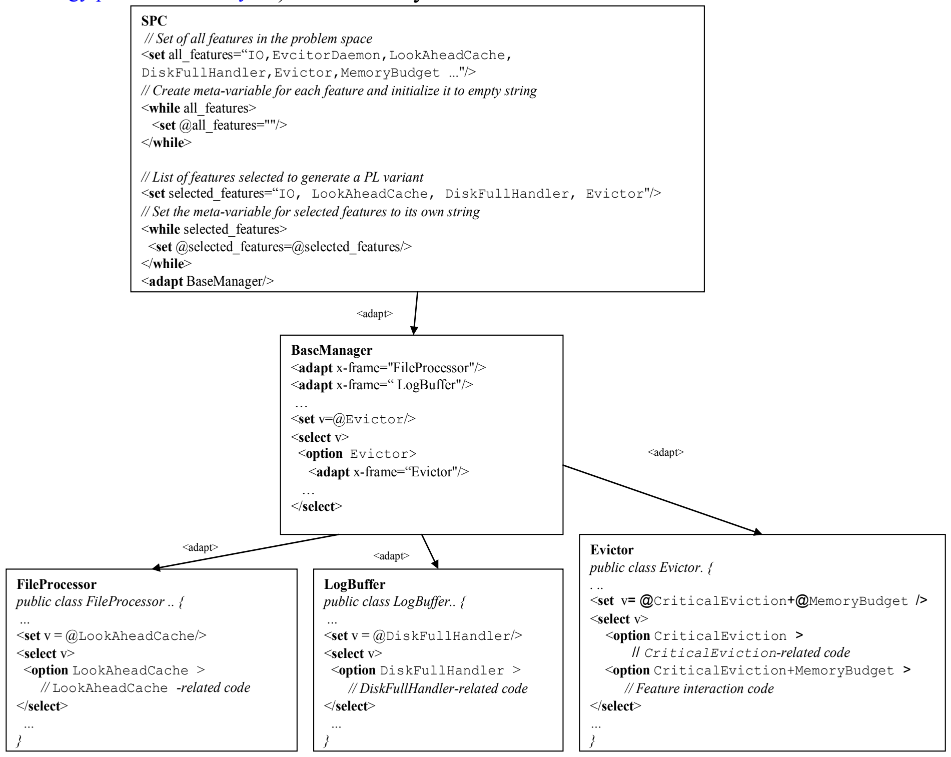
III. PREPROCESSING PROBLEMS IN BERKELEY DB

We use Berkeley DB to illustrate common preprocessing problems in handling product variants. 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. In that sense, Berkeley DB forms a Product Line, whose members implement different selections of features. Berkeley DB designers chose to use runtime mechanisms to accommodate required features into a custom DB system. In earlier studies, Berkeley DB was converted into a Product Line in which features were managed at the construction time (i.e., before execution) with AspectJ [12] and CIDE [5]. In our study, we also managed features at the design-time with a preprocessing notation described in Section II.

A. Overview of our study

The Berkeley DB consists of five subsystems, namely *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 a *persistence layer*. These five subsystems are designed with 232 files that form an architecture shared by all the DB system variants. Two of those files, namely **FileProcessor**, and **LogBuffer** are shown in Figure 2. In our experiment, we did not change the original design of the Berkeley DB.

38 features such as `IO`, `LookAheadCache` or `DiskFullHandler` can be optionally included into custom DB system variants. Berkeley DB designers used runtime mechanisms to configure features into custom DB products. For example, settings for `LookAheadCache` feature are stored in an environmental property file (called *je.properties*), and developers can change this file to include and re-configure the behavior of this feature.



```
SPC
 // Set of all features in the problem space
<set all_features="IO,EvcitorDaemon,LookAheadCache,
DiskFullHandler,Evictor,MemoryBudget …"/>
// Create meta-variable for each feature and initialize it to empty string
<while all_features>
  <set @all_features=""/>
</while>

// List of features selected to generate a PL variant
<set selected_features="IO, LookAheadCache, DiskFullHandler, Evictor"/>
// Set the meta-variable for selected features to its own string
<while selected_features>
 <set @selected_features=@selected_features/>
</while>
<adapt BaseManager/>
```

```
BaseManager
<adapt x-frame="FileProcessor"/>
<adapt x-frame=" LogBuffer"/>
 …
<set v=@Evictor/>
<select v>
 <option Evictor>
   <adapt x-frame="Evictor"/>
  …
</select>
```

```
FileProcessor
public class FileProcessor .. {
 …
<set v = @LookAheadCache/>
<select v>
 <option LookAheadCache >
    // LookAheadCache -related code
</select>
 …
}
```

```
LogBuffer
public class LogBuffer.. {
 …
<set v = @DiskFullHandler/>
<select v>
 <option DiskFullHandler >
    // DiskFullHandler-related code
</select>
 …
}
```

```
Evictor
public class Evictor. {
. ..
<set  v= @CriticalEviction+@MemoryBudget />
<select v>
   <option CriticalEviction >
       // CriticalEviction-related code
   <option CriticalEviction+MemoryBudget >
      // Feature interaction code
</select>
…
}
```

Figure 2. A preprocessing solution to managing features in Berkeley DB

In our study, we addressed 22 Berkeley DB features ranging from simple to complex. First, we analyzed the semantics of the features we decided to work with, and the way they affected the base DB code. We converted original runtime strategies for feature management to equivalent preprocessing strategies, in a similar way as Kastner et al. [12] converted them to AOP [16]. For that, we instrumented each base component affected by features with preprocessing commands to manage feature impact on the base code. Our preprocessing representation for Berkeley DB, shown in Figure 2, follows conventions described in Section II. Parameter *features_selected* in **SPC** specifies features to be included in a required DB variant, in case of our example, features `IO`, `LookAheadCache`, `DiskFullHandler` and others. **SPC** adapts **BaseManager**, which in turn adapts all DB base components. **BaseManager** plays the role of an integrator/composer of a custom DB variant in terms of its base components, propagating customizations for selected features to the base components. For clarity, in Figure 2, we have shown only two of such base components, namely **FileProcessor** and **LogBuffer**. In addition, **BaseManager** also <**adapt**>s feature-specific base components. This is because for some of the features, besides adding small fragments of code at variation points in base components upon feature selection, we must also add new files 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).

B. *Patterns of feature impact on DB base components*

Features that affect a small number of base components at a small number of variation points, and without interacting with other features are easily handled by preprocessing. Such features may require adding new classes to the DB base, e.g., features `Evictor` and `Statistics`, or adding member variables and methods to classes. Each such feature impact is handled by placing these member variables and methods within <**select-option**> block in affected base components.

Among 22 features in DB that we addressed in the experiment, five were preprocessing-friendly.

First type of complication occurs when the number of variation points at which features affect a given base component grows. The impact may come from one or more features. Then, the base code becomes densely populated with <**select**>s, which is one of the often mentioned drawbacks of using preprocessing to handle product variants. Among other examples, 12 features affect base component **Environmentimpl** inducing 38 variation points, and 10 feature affect file **FileManager**, inducing 40 variation points. The number of variation points per class ranges from 1 to 35, with average 5.72.

Further complications appear when the impact of one feature becomes scattered over many variation points, and spreads through many base components. For example, feature `MemoryBudget` affects 32 DB base components at a total of 190 variation points, `Statistics` (34 variation points), `CheckSum` and `Evictor` (each with 27 variation points), and `CriticalEviction` (23 variation points). When modifying scattered features, we must propagate changes to all the relevant variation points. Maintenance becomes difficult and error-prone.

Feature dependencies and interactions bring new dimension of difficulties for preprocessing. A feature f_1 depends on feature f_2 if f_1 refers to code of *f2*, or if the presence or absence of *f2* in a product variant affects implementation of *f1*.

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.

Figure 3 shows an example of feature interactions in which feature `CriticalEviction` interacts with feature `Evictor`, and feature `MemoryBudget` interacts with feature `CriticalEviction`. The net result of those interactions shows at variation points in file **Evictor**. If we select feature `CriticalEviction`, method *doCriticalEviction* must be included into the **Evictor**. If at the same time we select feature `MemoryBudget`, method *doCriticalEviction* must be modified with code related to that feature.

Feature code scattering and feature interactions lead to hidden dependencies among many variation points. These dependencies have to be understood to modify code without unwanted side-effects. Figure 3 shows an example of a hidden dependency induced by feature interactions. Features `CriticalEviction` and `MemoryBudget` affect the **Evictor**, which is only included if the `Evictor` feature is selected. Therefore, once the variation point that includes the **Evictor** is removed, all variation points introduced by the features `CriticalEviction` and `MemoryBudget` in file **Evictor** will be automatically affected.

```
Evictor
public class Evictor {
…
<set v= @CriticalEviction+@MemoryBudget />

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

Figure 3. Feature interactions

In Berkeley DB, we find 38 instances of feature interactions, scattered across 7 base components.

IV. XCPP: FINDING FEATURES WITH QUERIES

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 components, other features that a given feature interacts with, and the relevant variation points.

XCpp helps developers find feature code and analyze them in the many contexts where they occur. Developers specify features of interest in FQL (Feature Query Language). XCpp evaluates queries and displays the results.

In FQL, we can ask queries such as "which base components are affected by feature *f* and at which variation points?", "which features interact with feature *f*?", "in which base components and at which variation points feature f_1 interacts with feature f_2?". FQL is an SQL-like notation. We write queries in terms of elements meaningful at the preprocessing level such as base components and parameters of <**select-option**>.

Queries follow the format shown in Figure 4. Keywords are in **bold**. Clause **declare** introduces names of preprocessing elements which are used in the rest of the query. Clause **select** lists preprocessing elements - query results - we want to find. Clause **where** defines query conditions that constrain the result in terms of their participation in relationships and attribute values of preprocessing elements.

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

Figure 4. Format of feature queries

For example, query shown in Figure 5 finds all the code of feature `MemoryBudget`, and all base components affected by this feature. The two query conditions say that we are interested in all the base components 'x' that contain <**option**> labeled with feature name `MemoryBudget`. Since all the variation points at which `MemoryBudget` affects base components appear as <**option**>s marked with relevant feature names, this query effectively finds the desired result.

```
declare base_component x; option o;
select x, o
where o.f-names = "MemoryBudget"
        and Contains (x,o)
```

Figure 5. Finding feature code in base components

Finding and analysis of feature interactions is done in a similar way, using wild characters '*' to specify features of interest. Feature interactions occur at <**option**>s labeled with names of interacting features (separated by '+'). Query of Figure 6 finds all the variation points where features `CriticalEviction` and `MemoryBudget` interact one with another (the order in which features are listed is irrelevant).

```
declare base_component x, option o;
select x, o
where o.f-names="*CriticalEviction+MemoryBudget*"
    and Contains (x,o)
```

Figure 6. Finding feature interactions (1)

The two queries of Figure 7 find all the variation points where feature `Evictor` affects base components or interacts with other features. Queries return 28 variation points located in 12 base components.

```
declare base_component  x; option o;
select x, o
where o.f-names = "*Evictor*"
   and Contains (x,o)
```

```
declare base_component  x; option o;
select x, o
where o.f-names = "*"
      and Contains ("Evictor",o)
```

Figure 7. Finding feature interactions (2)

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 8 shows another useful query, which finds all the base components that adapt file `Evictor` directly or indirectly.

```
declare base_component x;
select x
where Adapts*(x, "Evictor")
```

Figure 8. Finding adapting base components

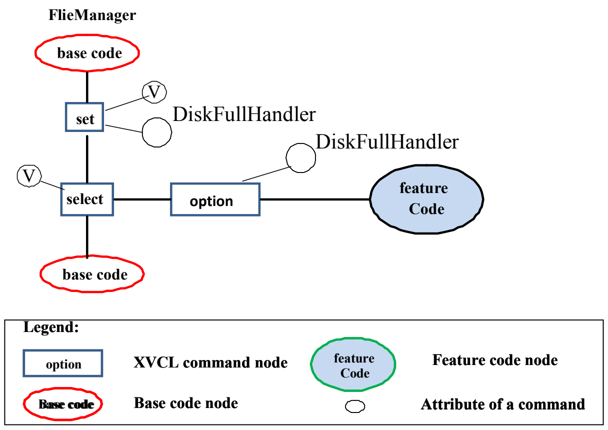


Figure 9. An AST of preprocessing information

To evaluate queries, we build Abstract Syntax Trees (AST) representing preprocessing information in base components. AST nodes correspond to preprocessing commands such as <**select**>, <**option**>, <**adapt**> or <**set**>. Attributes of command nodes contain variable names, their values and feature names at respective <**option**>s. Other AST nodes contain segments of base and feature code. Figure 9 shows an AST for the base component **FileManager**. XCpp's query evaluator traverses ASTs to find <**option**>s of interest.

Figure 10 shows the main components of our query system: the Feature Analyzer parses the base components and analyzes the feature code marked by XVCL commands embedded in base components. A base component Knowledge Base (FKB) stores all the information required to answer queries, collected by the Feature Analyzer. A user enters a query through a front-end tool. The Query Evaluator evaluates the query and fetches necessary information from the FKB via APIs. The query results are then presented to the user.

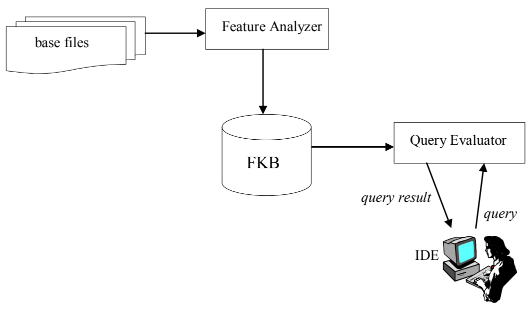


Figure 10. An overview of the feature query system

V. EVALUATION

Locating features with queries improves the readability of code instrumented with preprocessing directives. We can locate scattered features and analyze feature interactions. XCpp shows specific features at their variation points where they affect the base code or interact with other features, helping developers understand the scale of feature impact, and analyze feature code in multiple contexts at which they occur.

To trace the impact of changing features, developers must examine multiple variation points in multiple files, rapidly switching among various analysis contexts. XCpp helps in that. To illustrate this, suppose we want to change feature `MemoryBudget`. For that, we may need to switch among 190 variation points where `MemoryBudget` affects 32 base components. `MemoryBudget` also interacts with features `Evictor` and `CriticalEviction` and any change to `MemoryBudget` may affect these features. Therefore, 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.

TABLE 1 and TABLE 2 provide summaries of feature characteristics that affect feature maintenance effort. To enhance or fix an error in feature code, developers need examine variation points in base components affected by features (shown in TABLE 1), and variation points in interacting features (shown in TABLE 2). XCpp helps in finding and viewing all the variation points that need to be examined.

The above arguments hint at the potential gains in maintenance productivity due to XCpp's capability to find feature code in various contexts. It would be useful to evaluate the impact of XCpp on the maintenance/reuse effort in a controlled experiment. However, 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.

TABLE 1. FEATURE IMPACT ON BASE COMPONENTS

Feature involved in change	# base classes affected	# relevant variation points
MemoryBudget	32	190
Evictor	12	28
CheckSum	10	28
Statistics	10	34
CheckPointer	5	34
CpByteConfig	4	6
CpTimeConfig	4	7

TABLE 2. FEATURE INTERACTIONS

Feature involved in change	Interacting feature	# relevant variation points
CheckPointer	Statistics	22
MemoryBudget	Evictor	5
MemoryBudget	CriticalEviction	1
SyncIO	IO	4
EvictorDaemon	Evictor	3

While XCpp can ease feature understanding and maintenance, the essential difficulty of having to analyze feature code at many variation points and in multiple contexts still remains. All these contexts must be understood before implementing any changes. Furthermore, as it often happens, some changes should be propagated only to certain product variants, without affecting yet other product variants. Implementing such changes introduces more control parameters and more variation points in base components, which further complicates understanding, reusing and maintenance of the base components in preprocessing representation.

Despite these limitations, we believe feature analysis with XCpp improves usability of preprocessors, increasing their value as variation mechanisms in managing product variants.

We believe that in general, radical improvements of approaches that attempt to manage features in the base code in preprocessing style are difficult to achieve. As Karhinen et al. [3] 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 [7][2], we used generic design as means to manage features in SPLs. XVCL provides simple 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 (base components). 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 implement required selected features. This form of feature management and reuse has been introduced in Frame Technology™ [17]. XVCL, based on frame concepts, has been applied in lab studies [18][19] and industrial projects [20][21]. The approach is comprehensibly covered in [11].

VI. Related work

Initial ideas of feature management with queries and XCpp were reported in a short paper [27].

The many ways of using preprocessors and their common pitfalls have been widely discussed [1][3][4]. In this paper, we summarized problems reported by others, and illustrated them with examples from the Berkeley DB SPL.

Research community proposed Feature-Oriented Programming (FOP) [8] 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 SPLs. Mixin technique [22] has been widely used for FOP, with AHEAD [23] being its most advanced realization. AHEAD provides powerful solution for feature management in many situations, but may not be geared for fine-granularity impact features [5]. A number of authors also proposed Aspect-Oriented Programming (AOP) [16] to realize feature management in compositional way [24]. 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 [12].

In view of the above findings, Kastner et al. [5] 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. In contrast to CIDE, XCpp uses programming language-independent preprocessing-like program representation to achieve similar goals. XCpp represents preprocessing constructs as an AST, with segments of base code appearing at AST nodes. By promoting preprocessing to the first-class meta-level program constructs, XCpp can analyze and help one understand feature implementation and impacts.

In CIDE, annotations may be validated to preserve language rules, while XCpp <**select**> constructs may be placed in arbitrary program points, leading to syntactic errors in the generated code. Aligning XCpp representation with constructs of the underlying programming language is possible and can alleviate some problems, but aligning is not enforced by the method. In addition, in some cases XCpp deliberately breaks such alignment for better flexibility in feature management. Validating meta-level transformations is the strength of CIDE, while simplicity, language-independence and reliance on commonly used preprocessing are strengths of XCpp.

Reuse via software Product Line approach [7][2] uses architecture/component design as prime means to manage features. Still, complementary variation mechanisms such as preprocessing are needed to manage feature impact on components [13]. XCpp offers add-in value in this context.

VII. Conclusions

In addition to component/architecture design, preprocessors are often used to manage fine-grained features in software Product Lines. 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 find scattered feature code and feature interactions, improving readability of programs manipulated by preprocessors. XCpp applies queries to help developers navigate through feature-related code, showing 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 [7][2]. We believe XCpp contributes new values in this context.

In evaluation, we stress that XCpp avoids only accidental complexities of preprocessing (in Brooks' sense [25]). 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 are difficult to handle. In the paper, we pointed out the general nature of the difficulties of addressing these problems in the frame of approaches that rely on managing feature code in a base program.

Preprocessing directives can be viewed as kind of a meta-data that extends conventional programs by adding configuration knowledge to them. XCpp uses queries to analyze the configuration information. XCpp approach can be useful add-in technique for other systems that work with meta-data: Pragmas in Smalltalk contain extra information that can be interpreted by tools. Java and JEE annotations [26] contain meta-data that extends program behavior (for example, weaving AOP's advices can be expressed in some annotation systems). Annotations can be analyzed for understanding in XCpp-like fashion.

XCpp described in this paper can be improved by providing more synthetic, visual presentations of features under analysis. Integrating XCpp feature analysis with analysis of the base/feature code would be beneficial, but such a system would substantially diverge from preprocessors, making the solution language-dependent. Nevertheless, this is an interesting avenue to explore.

ACKNOWLEDGEMENT

This research is supported by the NUS research grant RP-252-000-336-112, the Chinese NSF grant 60703060, and the State 863 project 2007AA01Z480.

REFERENCES

[1] M. Ernst, G. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," IEEE Transactions on Software Engineering, Dec. 2002, pp. 1146-1170

[2] P. Clements, and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

[3] 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

[4] 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.

[5] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," Proc. Int. Conf. on Soft. Eng., ICSE'08, Leipzig, Germany, May 2008, pp. 311-320

[6] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake, "FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems" Software Engineering for Tailor-made Data Management 2008, pp. 1-6

[7] J. Bosch, Design and Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Welsey, 2000

[8] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects" European Conf. on Object-Oriented Programming, ECOOP'97, 1997, pp. 419-443

[9] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in Software Product Families: Problems and Issues during Product Derivation," Proc. Software Product Lines Conference, SPLC'04, Boston, MA., Aug. 2004, pp. 165-182

[10] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://xvcl.comp.nus.edu.sg

[11] S. Jarzabek, Effective Software Maintenance and Evolution: Reuse-based Approach, CRC Press Taylor & Francis, 2007

[12] 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

[13] 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

[14] J. Sun, H. Zhang, Y. Li, and H. Wang, "Formal Semantics and Verification for Feature Modeling," Proc. of 10th Int. Conf. on Engineering of Complex Computer Systems, ICECCS'05, Shanghai, June 2005. pp. 303-312

[15] H. Wang, Y. Li, J. Sun, H. Zhang, and J. Pan,"Verifying Feature Models using OWL," Journal of Web Semantics, Vol. 5 No. 2, June 2007, Elsevier, pp. 117-129.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," Europ. Conf. on Object-Oriented Programming, ECOOP'97, Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242

[17] P. Bassett, Framing software reuse - lessons from real world, Yourdon Press, Prentice Hall, 1997

[18] S. Jarzabek, and S. Li, "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," Proc. European Soft. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Soft. Eng., ESEC-FSE'03, Sept. 2003, Helsinki, pp. 237-246;

[19] J. Yang, and S. Jarzabek, "Applying a Generative Technique for Enhanced Reuse on JEE Platform," 4th Int. Conf. on Generative Programming and Component Engineering, GPCE'05, Sep 29 - Oct 1, 2005, Tallinn, pp.. 237-255

[20] U. Pettersson, and S. Jarzabek, "An Industrial Application of a Reuse Technique to a Web Portal Product Line," European Soft. Eng. Conf. and ACM SIGSOFT Symp. on the Foundations of Soft. Eng., ESEC-FSE'05, Sept. 2005, Lisbon, pp. 326-335

[21] W. Zhang, and S. Jarzabek, "Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices," 9th Int. Soft. Product Line Conf., SPLC'05, Sept. 2005, Rennes, France, pp. 57-69

[22] Y. Smaragdakis, and D. Batory, "Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," ACM TOSEM, Vol. 11, No. 2, 2002, pp. 215-255

[23] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," Proc. Int. Conf. on Soft. Eng.,, ICSE'03, Portland, Oregon, May 2003, pp. 187-197

[24] S. Apel, C. Kästner, T. Leich, and G. Saake, "Aspect Refinement – Unifying AOP and Stepwise Refinement." Journal of Object Technology, Vol. 6, No. 9, Special Issue. TOOLS EUROPE 2007, Oct. 2007, pp. 13–33

[25] F.P. Brooks, "No Silver Bullet," Comp. Mag., April 1986

[26] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[27] S. Jarzabek, H. Zhang, Y. Lee, Y. Xue, and N. Shaikh, "Increasing Usability of Preprocessing for Feature Management in Product Lines with Queries," Int. Conf. Software Engineering, ICSE'09, Vancouver, Canada, May 2009, pp. 111-114 (a track on New Ideas and Emerging Results).