

Virtual Separation of Concerns – A Second Chance for Preprocessors

Christian Kästner, School of Computer Science, University of Magdeburg, Germany

Sven Apel, Department of Informatics and Mathematics, University of Passau, Germany

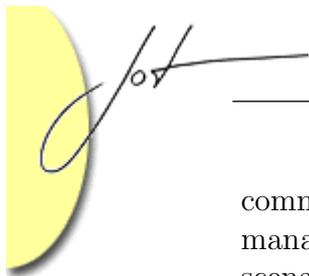
Conditional compilation with preprocessors like *cpp* is a simple but effective means to implement variability. By annotating code fragments with *#ifdef* and *#endif* directives, different program variants with or without these fragments can be created, which can be used (among others) to implement software product lines. Although, preprocessors are frequently used in practice, they are often criticized for their negative effect on code quality and maintainability. In contrast to modularized implementations, for example using components or aspects, preprocessors neglect separation of concerns, are prone to introduce subtle errors, can entirely obfuscate the source code, and limit reuse. Our aim is to rehabilitate the preprocessor by showing how simple tool support can address these problems and emulate some benefits of modularized implementations. At the same time we emphasize unique benefits of preprocessors, like simplicity and language independence. Although we do not have a definitive answer on how to implement variability, we want highlight opportunities to improve preprocessors and encourage research toward novel preprocessor-based approaches.

1 INTRODUCTION

The C preprocessor *cpp* [14] and similar tools¹ are broadly used in practice to implement variability. By annotating code fragments with *#ifdef* and *#endif* directives, these can later be excluded from compilation. With different compiler options, different program variants with or without these fragments can be created.

The usage of *#ifdef* and similar preprocessor directives has evolved into a common way to implement *software product lines (SPLs)*. A software product line is a set of related software systems (*variants*) in a single domain, generated from a

¹Historically, *cpp* has been designed for meta-programming. Of its three capabilities, file inclusion (*#include*), macros (*#define*), and conditional compilation (*#ifdef*), we focus only on conditional compilation, which is routinely used to implement variability. There are many preprocessors that provide similar facilities. For example, for Java ME, the preprocessors *Antenna* (<http://antenna.sf.net>) is often used; the developers of Java's Swing library developed their own preprocessor *Munge* (<http://weblogs.java.net/blog/tball/archive/munge/doc/Munge.html>); the languages Fortran and Erlang have their own preprocessors; some browsers support conditional compilation in JavaScript; and conditional compilation is a language feature in C#, Visual Basic, D, PL/SQL, Adobe Flex, and others.



common managed code base [3, 27]. For example, in the domain of embedded data management systems, different variants are needed depending on the application scenario: with or without transactions, with or without replication, with or without support for flash drives, with different power-saving algorithms, and so on [29, 30]. Variants of an SPL are distinguished in terms of *features* [2, 17], which are domain abstractions characterizing commonalities and differences between variants – in our example, transactions, replication, or flash support are features. A variant is specified by a feature selection, e.g., the data-management system with transactions but without flash support, and so on.

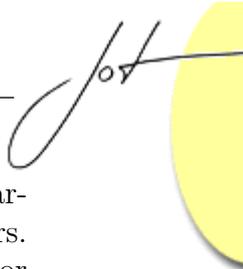
Preprocessors can be used to implement an SPL: Code that should only be included in certain variants, is annotated with `#ifdef X` and `#endif` preprocessor directives, in which *X* references a feature. Feature selections for different variants can be specified by using different configuration files or command line parameters as input for the compiler. Commercial product line tools like those from *pure::systems* or *BigLever* explicitly support preprocessors.

By this point, many readers may already object to preprocessor usage – and in fact, preprocessors are heavily criticized in literature as summarized in the claim “`#ifdef` Considered Harmful” [33]. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability [1, 11, 12, 24, 27, 33]. The use of `#ifdef` and similar directives breaks with the fundamentally accepted concept of *separation of concerns* and is prone to introduce subtle errors. Many academics recommend to limit or entirely abandon the use of preprocessors and instead implement SPLs with ‘modern’ implementation techniques that encapsulate features in some form of modules like components [27], framework/plugin architectures [16], feature modules [6, 28], aspects [23], and others.

Here, we take sides with preprocessors. We show how simple extensions of concepts and tools can avoid many pitfalls of preprocessor usage and we highlight some unique advantages over contemporary modularization techniques in the context of SPL development. Since we aim for separation of concerns without dividing feature-related code into physically separated modules, we name this approach *virtual separation of concerns*. We do not give a definitive answer on how to implement an SPL (actually, we are not sure ourselves and explore different paths in parallel), but we want to bring preprocessors back into the race and encourage research toward novel preprocessor-based approaches.

2 CRITICISM

Let us start with an overview of the four most common arguments against preprocessors: lack of separation of concerns, sensitivity to subtle errors, obfuscated source code, and lack of reuse.



Separation of concerns. Separation of concerns and related issues of modularity and traceability are usually regarded as the biggest problems of preprocessors. Instead of separating all code that implements a feature into a separate module (or file, class, package, etc.), a preprocessor-based implementation scatters feature code across the entire code base where it is entangled closely with the base code (which is always included) and the code of other features. Consider our data management example from the introduction: Code to implement transactions (acquire and release locks, commit and rollback changes) is scattered throughout the entire code base and tangled with code responsible for recovery and other features.

Lack of separation of concerns is held responsible for a lot of problems: To understand the behavior of a feature such as transactions or to remove a feature from the SPL, we need to search the entire code base instead of just looking into a single module. There is no direct traceability from a feature as domain concept to its implementation. Tangled code of other features distracts the programmer in the search. Tangled code is also a challenge for distributed development because developers working on different concerns have to edit the same files. Scattered code furthermore affects program comprehension negatively and, consequently, reduces maintainability of the source code. Scattering and tangling feature code is contrary to decades of software engineering education.

Sensitivity to subtle errors. Using preprocessors to implement optional features can easily introduce errors on different levels that can be very difficult to detect. This already begins with simple syntax errors. Preprocessors such as *cpp* operate at the level of characters or tokens, without interpreting the underlying code. Thus, developers are prone to simple errors like annotating a closing bracket but not the opening one as illustrated in the code excerpt from Oracle's Berkeley DB² in Figure 1 (the opening bracket in Line 4 is closed in Line 17 only when feature *HAVE.QUEUE* is selected). We introduced this error deliberately, but such errors can easily occur in practice and are difficult to detect. The scattered nature of feature implementations intensifies this problem. The worst part is that compilers cannot detect such syntax errors, unless the developer (or customer) eventually builds a variant with a problematic feature combination (without *HAVE.QUEUE* in our case). However, since there are so many potential variants (2^n variants for n independent optional features), we might not compile variants with a problematic feature combination during initial development. Simply compiling *all* variants is also not feasible due to their high number, so, even simple syntax errors might go undetected for a long time. The bottom line is that errors are found only late in the development cycle, when they are more expensive to fix.

Beyond syntax errors, also type and behavioral errors can occur. When a developer annotates a method as belonging to a feature, she must ensure that the method is not called in a variant without the feature. For example, in Figure 2, method *set* should not be included in a read-only database, however in such variant a type error

²<http://www.oracle.com/technology/products/berkeley-db/>

```

1 static int __rep_queue_filedone(dbenv, rep, rfp)
2     DB_ENV *dbenv;
3     REP *rep;
4     __rep_fileinfo_args *rfp; {
5 #ifndef HAVE_QUEUE
6     COMPQUIET(rep, NULL);
7     COMPQUIET(rfp, NULL);
8     return (__db_no_queue_am(dbenv));
9 #else
10    db_pgno_t first, last;
11    u_int32_t flags;
12    int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14    DB_MSGBUF mb;
15 #endif
16    // over 100 further lines of C code
17 }
18 #endif

```

Figure 1: Code excerpt of Oracle’s Berkeley DB with a deliberately introduced syntax error in variants without *HAVE_QUEUE*.

```

1 class Database {
2     Storage storage;
3     void insert(Object key, Object data, Transaction txn) {
4         storage.set(key, data, txn.getLock());
5     }
6 }
7 class Storage {
8 #ifndef WRITE
9     boolean set(Object key, Object data, Lock lock) { ... }
10 #endif
11 }

```

Figure 2: Code excerpt with type error when feature WRITE is not selected.

will occur in Line 3 since the removed method *set* is still referenced. Even though compilers for statically typed languages can detect such problems, again this is only noticed when the problematic feature combination is eventually compiled. Worse of all are behavioral errors, for example annotating only to *release lock* call but forgetting the *acquire lock* call in some method, which is only noticed in some variants as a deadlock at runtime. Tests or common formal specification and verification approaches can be used to detect behavioral errors, but again this requires to check every variant for every feature combination.

Obfuscated source code. When implementing features with *cpp* or similar tools, preprocessor directives and statements of the host language are intermixed in the same file. When reading source code, many *#ifdef* and *#endif* directives distract from the actual code and can destroy the code layout (with *cpp*, every directive must be placed in its own line). There are cases where preprocessor directives entirely obfuscate the source code as illustrated in Figure 3, leading to code that is hard to read and hard to maintain.

```
1 class Stack {
2     void push(Object o
3 #ifdef TXN
4     , Transaction txn
5 #endif
6     ) {
7         if (o==null
8 #ifdef TXN
9             || txn==null
10 #endif
11         ) return;
12 #ifdef TXN
13         Lock l=txn.lock(o);
14 #endif
15         elementData[size++] = o;
16 #ifdef TXN
17         l.unlock();
18 #endif
19         fireStackChanged();
20     }
21 }
```

Figure 3: Java code obfuscated by fine-grained annotations with *cpp*.

In Figure 3, preprocessor directives are used at a fine granularity [19], annotating not only statements but also parameters and part of expressions. We need to add eight additional lines just for preprocessor directives. Together with additional necessary line breaks, we need 21 instead of 9 lines for this code fragment. Furthermore, nested preprocessor directives and multiple directives belonging to different features as in Figure 1 are other typical causes of obfuscated code.

Although our example in Figure 3 appears extreme at first, similar code fragments can be found in practice. For example, in Figure 4, we illustrate the amount of preprocessor directives in *Femto OS*³, a small real-time operating system.

Lack of reuse. Finally, preprocessor usage restricts reuse. In contrast to components, which encapsulate code that can be reused in other projects (even outside the SPL), scattered feature code is usually aligned exactly for the current SPL. There is typically no abstraction or encapsulation.

For example, in our data management example, code to access flash memory may be scattered across the entire implementation. When flash memory access is also needed in another system, say an embedded operating system, we cannot simply reuse the scattered implementation by including a file or library. We need to extract and copy the code in our new project and thus maintain scattered and *replicated* code. Of course, also with preprocessors it is possible to modularize all code for flash memory access in a module or library, but (unless reuse is planned ahead) there is no incentive when developers grow accustomed to the easier form of scattered implementations.

³<http://www.femtoos.org/>

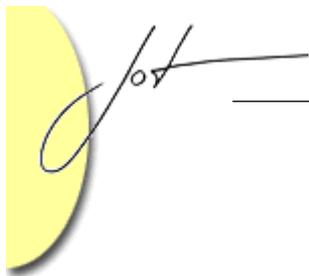


Figure 4: Preprocessor directives in the code of Femto OS: Black lines represent preprocessor directives such as *#ifdef*, white lines represent the remaining C code, comment lines are not shown.



3 PHYSICAL SEPARATION OF CONCERNS

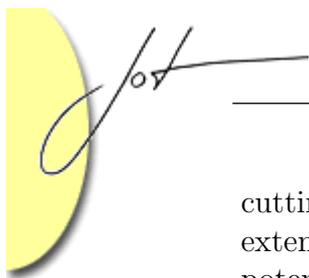
Before we discuss how we can improve preprocessors, let us have a look at the competitors. Specifically, we survey three implementation strategies that (from our perception) attract most research: components, frameworks, and modern module systems. They all have in common that they decompose the source code and implement each feature in a distinct module, thus they *physically* separate concerns.

Components. Apart from preprocessors, one of the most common approaches to implement SPLs is to build components. When designing an SPL, developers first identify common and variable parts and introduce a component architecture. Parts of the system that correspond to features are modularized and implemented as reusable components. The advantage of components is that all parts are implemented modularly: Implementations are hidden behind interfaces and ideally features can be developed, understood, and maintained in isolation.

To build a variant for a given feature selection, a developer reuses the SPL's components and integrates them into the final product. To this end, the developer typically implements some glue code to fit the components together. There is no full automation such that we could automatically generate a program for a feature selection. While this approach has proved to be practical in industry, there are issues. The smaller the components are, the more glue code and thus development effort is required for deriving a variant. Generally, components are useful for coarse grained features like receivers and decoders in the home entertainment market, but are challenged for SPLs with a high number of fine-grained features. Finally, also crosscutting features challenge the modularization of components. If a feature like transactions affects multiple parts of the system (and would lead to a high degree of scattering in a preprocessor-based implementation), it is difficult to modularize, and much glue code is needed to connect such module to the remaining system.

Frameworks. Framework and plug-in architectures are similar to components, but aim for automation. The main difference is that components are not assembled with glue code as needed, but that already a common framework exists in which features are *plugged in*. That is, a framework exhibits an extension point, which is extended by one or more plug-ins (features). A framework can be executed with or without plug-ins. We can automatically generate a variant for a feature selection by assembling the corresponding plug-ins without further development effort. Like components, plug-ins are ideally self-contained modules, thus achieving physical separation of concerns.

Still, regarding granularity and crosscutting features, frameworks exhibit problems similar to those of components. If an SPL has many fine-grained features (which would lead to a high degree of scattering in a preprocessor-based implementation), the framework becomes very complex and difficult to understand. Cross-



cutting features are challenging, because the framework must provide many small extension points (e.g., all points at which locks for the transaction mechanism are potentially acquired or released). Thus, part of a feature's implementation can become a mandatory part of the framework which contradicts the desired separation of concerns to some degree.

Modern module systems. In the last decade, researchers have invested immense efforts into developing new programming language concepts to modularize crosscutting implementations. Concepts like aspect-oriented programming [23], feature-oriented programming [28], multi-dimensional separation of concerns [35], virtual classes [26], mixin layers [32], classboxes [8], and many more, have been proposed to separate crosscutting concerns. For example, the entire (otherwise scattered) implementation of a feature can be encapsulated in an aspect, which describes where and how the behavior of the base program must be changed (e.g., acquire and release locks). That is, in contrast to preprocessors, all code of this feature is modularized. In these approaches, modules are typically composed with a specialized compiler; variants are generated by deciding which modules to compile into the program.

Applicability of such language extensions to SPL development has been shown in a number of academic case studies, however, so far, they had little influence on industrial practice. One of the reasons is that all these approaches introduce new language concepts. Developers need to learn new languages and to think in new ways. Their effect on program comprehension has still to be evaluated. Furthermore, in contrast to preprocessors, which are usually language-independent, an extended language must be provided for every language that is used in an SPL (e.g., AspectJ for Java, AspectC for C, Aspect-UML for UML, AspectXML for XML). Most languages are experimental and do not provide the tool support to which developers have grown accustomed with modern IDEs as Visual Studio or Eclipse.

Special Challenge: Optional Feature Problem. There is a special problem with which all approaches that modularize features struggle: Features are not always independent from each other and there is often code that belongs not only to a single feature, but that connects multiple features [21,25].

Consider the standard expression problem [38]: We have an evaluator of mathematical expressions and want to be able to add new operations to our expressions (*evaluate*, *print*, *simplify*, ...). At the same time, we want to be able to add new kinds of expressions (*plus*, *power*, *ln*, ...). The implementation of *evaluate a plus expression* (e.g., $3 + 1 = 4$) concerns both feature *plus* and feature *evaluate*. If *evaluate* is not selected, this code is not needed; if *plus* is not selected, this code is not needed either. But how can we modularize code such that we can freely select features from both operations and expressions?

In Figure 5 (a) and (b) you see the two standard forms of modularization, we either modularize expressions or operations. Thus, in Figure 5 (a), we can easily

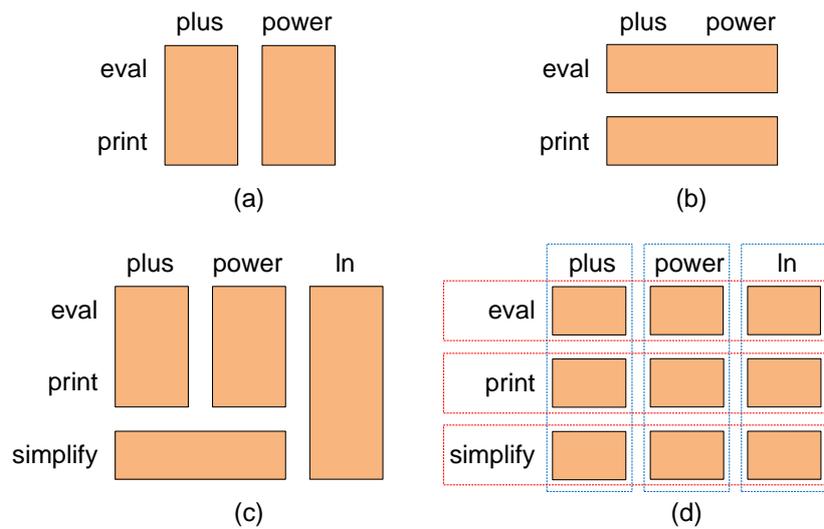
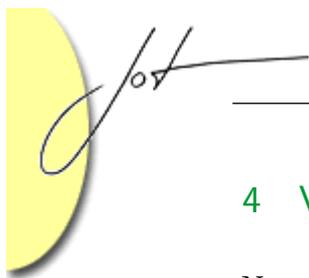


Figure 5: Modularization of interacting features: (a) modularized by expressions; (b) modularized by operations; (c) modularized by expressions, then extended twice; (d) small modules grouped by data types and operations.

remove or add expressions but not operations, and in Figure 5 (b), we can remove and add operations but not expressions. Researchers have found advanced solutions of the expression problem (e.g., using generics or aspects) to extend the code with a new optional feature, independent of what modularization has been used initially. As visualized in Figure 5 (c), we can add a new module *simplify* without changing existing modules, and then add a new module *ln*, without changing existing modules. But still, we cannot mix and match features freely but create very specific constraints instead.

The solution to this problem (described in different contexts as *lifters* [28], *origami* [5], or *derivatives* [25]) is to break down these modules into smaller modules and group them back again. The small modules may belong to multiple features. This is illustrated in Figure 5 (d), in which the code that implements evaluating a plus expression is encapsulated in its own module (top-left) and belongs to both features *simplify* and *plus* (indicated by dotted lines).

Splitting a program into too many small modules can be problematic. As described above, some implementation approaches do not perform well with fine-grained modules (e.g., high amount of glue code or high number of extension points). Furthermore, although concerns have been separated, the developer who wants to understand a feature in its entirety (e.g., the entire *simplify* mechanism or the entire transaction subsystem) has to look into many modules and reconstruct the behavior in her mind. That is, we lose the benefits of traceability and modular reasoning for which we physically separated concerns in the first place.



4 VIRTUAL SEPARATION OF CONCERNS

Now, let us come back to preprocessors and how they can be improved. We address the four main problems listed in Section 2 and show how simple mechanisms and/or tool support can alleviate or solve them. Although we cannot claim to eliminate all disadvantages, we conclude this section by pointing out some new opportunities and unique advantages that preprocessors offer.

Separation of Concerns

One of the key motivations of modularizing features is that developers can find all code of a feature in one spot and reason about it without being distracted by other concerns. Clearly, a scattered, preprocessor-based implementation does not support this kind of lookup and reasoning, but the core question “what code belongs to this feature” can still be answered by tool support in the form of *views* [15, 22, 31].

With relatively simple tool support, it is possible to create an (editable) view on the source code by hiding all irrelevant code of other features (technically this can be implemented like code folding in modern IDEs).⁴ In Figure 6, we show an example of a code fragment and a view on its feature *Transaction*. Note, we cannot simply remove everything that is not annotated by *#ifdef* directives, because we could end up with completely unrelated statements. Instead, we need to provide some context, e.g., in which class and method is this statement located; in Figure 6 we print the context information in gray and italic font. Interestingly, similar context information is also present in modularized implementations in the form of extension points and interfaces.

So, with simple tool support for providing views, we can emulate some advantages of physically separated features. Note, these views naturally emulate the ‘modularization’ of the expression problem [15], the ‘evaluate plus’ code simply occurs in both the views on feature *evaluate* and feature *plus*.

Beyond views on individual features, (editable) views on variants are possible [13, 22]. That is, a tool can show the source code that would be generated for a given feature selection and hide all remaining code of unselected features. With such a view, a developer can explore the behavior of a variant when multiple features interact, without distracting code of unrelated features. This goes beyond the power of physical separation, with which the developer has to reconstruct the behavior of multiple components/plugin-aspects in her mind. Especially, when many fine-grained features interact, from our experience, views can be a tremendous help.

⁴Although editable views are harder to implement than read-only views, they are more useful since users do not have to go back to the original code to make a modification. Implementation of editable views have been discussed intensively in work on database or model roundtrip engineering. Furthermore, a simple but effective solution, which we apply in our tools is to leave a marker indicating hidden code [19]. Thus, modifications occur before or after the marker and can be unambiguously propagated to the original location.



```

1 class Stack implements IStack {
2     void push(Object o) {
3 #ifdef TXN
4         Lock l = lock(o);
5 #endif
6 #ifdef UNDO
7         last = elementData[size];
8 #endif
9         elementData[size++] = o;
10 #ifdef TXN
11         l.unlock();
12 #endif
13         fireStackChanged();
14     }
15 #ifdef TXN
16     Lock lock(Object o) {
17         return
18             LockMgr.lockObject(o);
19 #endif
20     ...
21 }

```

(a) original

```

1 class Stack {
2     void push() {
3         Lock l = lock(o);
4         l.unlock();
5     }
6     Lock lock(Object o) {
7         return
8             LockMgr.lockObject(o);
9     }
10 }

```

(b) view on TXN (hidden code is indicated by '[]', necessary context information is printed italic and gray)

Figure 6: View emulates separation of concerns.

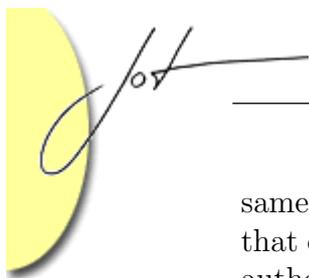
Nevertheless, some desirable such as separate compilation or modular type checking cannot be achieved with views.

Sensitivity to subtle errors

Also various kinds of errors that can easily occur with *#ifdef* annotations can be detected by adding tool support. In this section, we show how disciplined annotations can help [19,20] regarding syntax errors, such as the bracket mismatch in Figure 1 and how new product-line-aware type systems can help regarding type errors, such as calling an annotated method [10,18]. We do not focus on semantic errors like deadlocks, because they are not a specific problem of annotations but can occur equally in physically separated code.

Disciplined annotations are an approach to limit the expressive power of annotations in order to prevent syntax errors, without restricting the preprocessor's applicability to practical problems. Syntax errors arise from preprocessor usage that considers a source file as plain text, in which every character or token (including individual brackets) can be annotated. A safer way to annotate code is to consider the underlying structure of the code and allow programmers to annotate (and thus remove) only program elements like classes, methods, or statements. This way, syntax errors as in Figure 1 cannot occur.

Disciplined annotations may require more effort from developers, since only annotations based on the underlying structure are allowed. For some annotations that only worked on plain text with *cpp*, workarounds are required to implement the



same behavior with disciplined annotations. However, several authors have argued that disciplined annotations do not impose significant problems; even with *cpp* most authors strive for disciplined annotations anyway and consider anything else a ‘hack’ (see for example [7, 37]). We found that how to change undisciplined annotations to disciplined ones is typically obvious and follows simple patterns. Despite some necessary workarounds, disciplined annotations are still easier to use and more expressive for fine-grained extensions than components/plugin-ins/aspects for physical separation, which require to restructure source code entirely [19].

Technically, disciplined annotations require more elaborate tools, which have a basic understanding of the underlying artifacts. Such tools check whether annotations with a traditional preprocessor are in a disciplined form (this is equivalent to physical separation approaches in which each module can be checked for syntax errors in isolation). Alternatively, there are tools like CIDE [19] that manage annotations and ensure that only structural elements can be annotated in the first place. It has been shown that tools for disciplined annotations can be rapidly extended to different languages by generating parsers from existing grammar specifications [20].

Product-line-aware type systems can check that all variants in the product line are well-typed (i.e., can be compiled). The most important problems that can be detected this way are methods or types that are removed in some variants but still referenced, like in Figure 2 (problems that are less common in physical separation approaches since often common interfaces and separate compilation are used).

The basic idea of a product-line-aware type system is not only to check all method invocations during compilation, but to check whether each method invocation can be resolved *in every variant*. If both, reference and target are annotated with the same feature, the reference can be resolved in every variant, otherwise we have to check the relationship between both annotations. If there is any variant in which the target but not the reference is removed (as in Figure 2), the type system issues an error.⁵ That is, the entire SPL is checked in a single step by comparing annotations of all invocations and their respective targets, instead of checking every variant in isolation.

Type checking annotations again emulates some form of modules and dependencies between them. So instead of specifying that component *Transaction* imports component *Recovery*, we check these dependencies in scattered code using relationships between features in an SPL like ‘*selecting feature Transaction always implies selecting feature Recovery*’.

With disciplined annotations and product-line-aware type systems, we can bring preprocessors at least to the same level as physical separation approaches regarding error detection. Specifically product-line-aware type systems have been shown useful, so they have been adapted for several physical separation approaches as well

⁵There are many ways to describe and reason about relationships between features in an SPL, but their description is beyond the scope of this paper. Feature models and propositional formulas are common [4].



(e.g. [36]). Regarding the hardest part, semantic errors (i.e., incorrect behavior in some variants), both virtual and physical separation are on the same level. There are several approaches for SPL testing and applying formal methods, but they have similar problems (especially regarding scalability) independent of the implementation approach.

Obfuscated source code

When many annotations are used in the same file, it may be difficult to read the code, as illustrated in Figures 3 and 4. Preprocessors like *cpp* require two extra lines for each annotated code fragment (*#ifdef* and *#endif* both defined in their own line).

There are several ways how the representation can be improved. First, textual annotations with a less verbose syntax that can be used within a single line could help, and can be used with many tools. Second, views can help to focus on the relevant code, as discussed above. Third, visual means can be used to differentiate annotations from source code: Like some IDEs for PHP use different font styles or background colors to emphasize the difference between HTML and PHP in a single file, different graphical means can be used to highlight preprocessor directives. Finally, it is possible to eliminate textual annotations altogether and use the representation layer to convey annotations, as we show next.

In CIDE, textual annotations are abandoned; the tool uses background colors to represent annotations [19]. For example, all code belonging to the feature *Transaction* is highlighted with red background color. Using the representation layer, also our example from Figure 3 is much shorter as shown in Figure 7. Using background colors mimics our initial steps to mark features on printouts with colored text markers and can easily be implemented since the background color is not yet used in most IDEs. Instead of background colors the tool Spotlight uses colored lines next to the source code [9]. Background colors and lines are especially helpful for long and nested annotations, which may otherwise be hard to track. We are aware of some potential problems of using colors (e.g., humans are only able to distinguish a certain number of colors), but still, there are many interesting possibilities to explore.

Despite all visual enhancements, there is one important lesson: Using preprocessors does not require modularity to be dropped at all, but rather frees programmers from the burden of forcing them to physically modularize everything. Typically, most of a feature's code will be still implemented by a number of modules or classes, but calls may be scattered in the remaining implementation as necessary. In our experience from using CIDE, on a single page of code there are rarely annotations from more than two or three features.

```

1 class Stack {
2     void push(Object o, Transaction txn) {
3         if (o==null || txn==null) return;
4         Lock l=txn.lock(o);
5         elementData[size++] = o;
6         l.unlock();
7         fireStackChanged();
8     }
9 }

```

Figure 7: Annotated code represented by background color instead of textual annotation.

Lack of reuse

A scattered, annotated implementation cannot simply be reused in a different project. However, the core code of the feature’s implementation (e.g., the core locking mechanisms and rollback facility of the transaction feature) can often be easily reused, while only the invocations (the integration into the behavior of the system, e.g., calling lock and unlock) remain scattered. However, these scattered invocations would be difficult to reuse for another system outside the SPL anyway, also in a physical separated implementation.

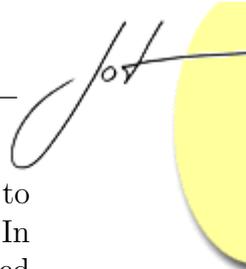
Nevertheless, there are two more complicated cases. Inherently crosscutting implementations often resist modularization. Although they can be modularized with modern approaches like aspect-oriented programming, aspect reuse in a different context is still difficult except for some simple homogeneous crosscutting concerns like tracing or profiling [34]. We conjecture that the effort necessary to reuse more complex aspects (e.g., implement numerous abstract pointcuts) is similar to the effort for adding scattered calls.

We recommend to follow the simple guideline “modularize feature code as far as possible, scatter remaining invocations”. This guideline is best practice anyhow, but easily ignored when developers grow accustomed to preprocessors. We argue that reuse of annotated code in different projects is not more difficult than reusing a physically separated implementation.

Unique advantages of preprocessors

In the previous section, we have shown how simple tool support can address most of problems commonly attributed to preprocessors. Although preprocessors can only *emulate* certain benefits of physically separated implementations, we argue that they are worth at least further consideration and evaluation. For those still not convinced, we present some distinct advantages of preprocessors over physically separated implementations in this section.

Preprocessors have a *very simple programming model*: Code can be annotated



and removed. Preprocessors are very easy to use and understand. In contrast to physical separation, no new languages, tools, or processes have to be learned. In many languages, preprocessors are already included, otherwise they can be added with lightweight tools. This is the main advantage of preprocessors which drives professionals to still use them despite all disadvantages.

Most preprocessors are *language independent* and provide a *uniform experience* when annotating different artifact types. For example, *cpp* can not only be used on C code but also also on Java code or HTML files. Instead of providing a tool or model for every language, each with different mechanisms (e.g., AspectJ for Java, AspectC for C, Aspect-UML for UML)⁶, preprocessors add the same simple model to *all* languages. Even with disciplined annotations (see above), a uniform experience can be achieved for multiple languages.

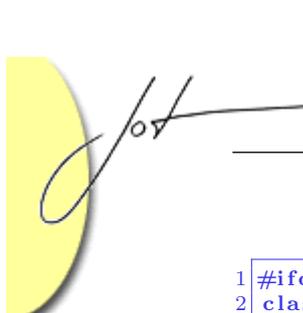
A (dominant) decomposition is still possible. Annotating code does not prohibit traditional means of separation of concerns. In fact, as discussed above, it is reasonable to still decompose the system into modules and classes and use preprocessors only where necessary. Preprocessors only add additional expressiveness, where traditional modularization techniques come to their limits regarding crosscutting concerns or multi-dimensional separation of concerns.

Finally, preprocessors can *handle multiple interacting optional features and shared code naturally*. Instead of being forced to create many additional modules, nested annotations provide an intuitive mechanism to include code only when two or more features are selected. In Figure 8, we show the annotation-based implementation of the expression problem (cf. Sec. 3). From this example, we can select every feature combination and can create all variants, without splitting the features into many small modules. In this scenario views on the source code, as described above, play to their strength.

5 CONCLUSION

We have argued that preprocessors are not beyond hope in addressing key problems of software product line development. With little tool support, we can address many problems on which preprocessors are often criticized. Views on the source code emulate modularity and separation of concerns; disciplined annotations and product-line-aware type systems detect implementation errors; editors can distinguish the difference between source code and annotations or even lift annotations to the representation layer; and with a little discipline from developers, also reuse can be achieved similar to approaches that modularize feature code. Together, we name these efforts *virtual separation of concerns* because, even though features are

⁶Also for physical separation, there is research that aims at simple and/or language-independent models and tools, e.g. [6]. These approaches often trade simplicity or generality for expressiveness, therefore they sometimes sacrifice benefits like separate compilation or type checking. A comprehensive discussion is outside the scope of this paper.



```

1 #ifndef ADD
2 class Add extends Expr {
3     Expr left, right;
4     Add(Expr l, Expr r)
5         { left=l; right=r; }
6 #ifndef EVAL
7     double eval() {
8         return left.eval() +
9             right.eval();
10    }
11 #endif
12 #ifndef PRINT
13     void print() {
14         left.print();
15         System.out.print("+");
16         right.print();
17     }
18 #endif
19 }
20 #endif

21 #ifndef POWER
22 class Pow extends Expr {
23     Expr base, exp;
24     Pow(Expr b, Expr e)
25         { base=b; exp=e; }
26 #ifndef EVAL
27     double eval() {
28         return
29             Math.pow(base.eval(),
30                 right.eval());
31     }
32 #endif
33 #ifndef PRINT
34     void print() {
35         left.print();
36         System.out.print("^");
37         right.print();
38     }
39 #endif
40 #endif

```

Figure 8: Preprocessor-based implementation of the expression problem (excerpt).

not physically separated into modules, this separation is emulated by tools.

We argue that *tool support is the key* to SPL development. For virtual separation it is essential to counter the problem of naive preprocessors. But also for physical separation, tool support for navigating between modules or showing how modules relate is important, especially when many small modules are required as for the optional feature problem (see Fig. 5).

While we do not eliminate all problems of preprocessors (for example, separate compilation is still not possible), preprocessors also have some distinct advantages like ease of use and language independence. Additionally, they provide a new perspective on the problem of multi-dimensional separation of concerns and optional interacting features.

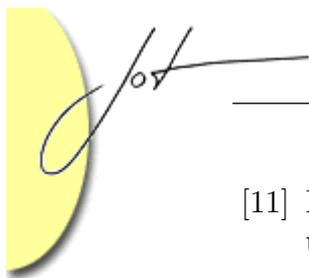
We do not have a definitive answer whether physical or virtual separation of concerns is better (and this depends very much on what you measure). We are still investigating both approaches in parallel, and have a look at a possible integration. With this paper, we want to encourage researchers to overcome their prejudices (usually from experience with *c++*) and to consider annotation-based implementations. At the same time, we want to encourage current practitioners that are currently using preprocessors to look for improvements. Since tool support is necessary for SPL implementation anyway, it is well worth investing also into tool support for new preprocessors and virtual separation of concerns. *Give preprocessors a second chance!*



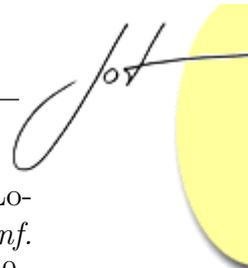
Acknowledgments We thank Jörg Liebig and Don Batory for helpful comments on earlier drafts of this paper. Furthermore, we thank Marko Rosenmüller and Jörg Liebig for the examples from Berkeley DB and Femto OS. Apel's work is supported in part by DFG project #AP 206/2-1.

REFERENCES

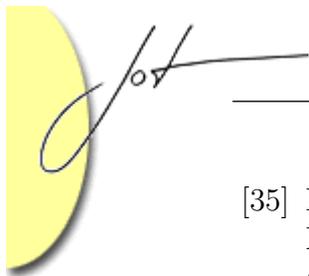
- [1] B. Adams, B. Van Rompaey, C. Gibbs, and Y. Coady. Aspect mining in the presence of the C preprocessor. In *Proc. AOSD Workshop on Linking Aspect Technology and Evolution (LATE)*, pages 1–6, New York, NY, USA, 2008. ACM Press.
- [2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009.
- [3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, MA, USA, 1998.
- [4] D. Batory. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20, Berlin/Heidelberg, Sept. 2005. Springer-Verlag.
- [5] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 81–92, Washington, DC, USA, 2002. IEEE Computer Society.
- [6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.
- [7] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290, Washington, DC, USA, 2001. IEEE Computer Society.
- [8] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 177–189, New York, NY, USA, 2005. ACM Press.
- [9] D. Coppit, R. Painter, and M. Revelle. Spotlight: A prototype tool for software plans. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 754–757, Washington, DC, USA, 2007. IEEE Computer Society.
- [10] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA, 2006. ACM Press.



- [11] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.
- [12] J. Favre. Understanding-in-the-large. In *Proc. Int’l Workshop on Program Comprehension*, page 29, Los Alamitos, CA, USA, 1997. IEEE Computer Society.
- [13] F. Heidenreich, I. Şavga, and C. Wende. On controlled visualisations in software product line engineering. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 303–313, Limerick, Ireland, Sept. 2008. Lero.
- [14] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, Dec. 1999.
- [15] D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 195–218. Springer-Verlag, 2004.
- [16] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP)*, 1(2):22–35, 1988.
- [17] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Nov. 1990.
- [18] C. Kästner and S. Apel. Type-checking software product lines – A formal approach. In *Proc. Int’l Conf. Automated Software Engineering (ASE)*, pages 258–267, Los Alamitos, CA, USA, Sept. 2008. IEEE Computer Society.
- [19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int’l Conf. Software Engineering (ICSE)*, pages 311–320, New York, NY, USA, May 2008. ACM Press.
- [20] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proc. Int’l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *LNBIP*, pages 175–194, Berlin/Heidelberg, June 2009. Springer-Verlag.
- [21] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int’l Software Product Line Conference (SPLC)*. SEI, Aug. 2009.
- [22] C. Kästner, S. Trujillo, and S. Apel. Visualizing software product line variabilities in source code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, Limerick, Ireland, Sept. 2008. Lero.



- [23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin. Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242, Berlin/Heidelberg, July 1997. Springer-Verlag.
- [24] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer Society.
- [25] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121, New York, NY, 2006. ACM Press.
- [26] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 90–99, New York, NY, USA, 2003. ACM Press.
- [27] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Secaucus, NJ, USA, 2005.
- [28] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Berlin/Heidelberg, June 1997. Springer-Verlag.
- [29] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 2009. accepted for publication.
- [30] M. Seltzer. Beyond relational databases. *Commun. ACM*, 51(7):52–58, 2008.
- [31] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9, New York, NY, USA, 2007. ACM Press.
- [32] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
- [33] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198, Summer 1992.
- [34] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 481–497, New York, NY, USA, 2006. ACM Press.



- [35] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society.
- [36] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA, 2007. ACM Press.
- [37] M. Vittek. Refactoring browser with preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
- [38] P. Wadler et al. The expression problem. Discussion on the Java-Genericity mailing list, 1998.

ABOUT THE AUTHORS



Christian Kästner is a Ph. D. student in Computer Science at the University of Magdeburg, Germany. His research interests include languages and tools for software product lines and (virtual) separation of concerns. He can be reached at kaestner@iti.cs.uni-magdeburg.de. See also <http://wwwiti.cs.uni-magdeburg.de/~ckaestne/>.



Sven Apel is a post-doctoral associate at the Chair of Programming at the University of Passau, Germany. He received a Ph. D. in Computer Science from the University of Magdeburg, Germany in 2007. His research interests include advanced programming paradigms, software product lines, and algebra for software construction. He can be reached at apel@uni-passau.de. See also <http://www.infosun.fim.uni-passau.de/cl/staff/apel/>.