it 1/2012

Virtual Separation of Concerns: Toward Preprocessors 2.0[1]

Virtuelle Trennung von Belangen: Ein Schritt zu Präprozessoren 2.0

Christian Kästner, Laureate of the GI Dissertation Award 2010[2], Philipps University Marburg

Summary 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. We show how simple tool support – views, visualizations, disciplined annotations, and variability-aware type systems – can address these problems and emulate some benefits of modularized implementations. Instead of separating source code into files, we pursue a "virtual separation of concerns".

►►► **Zusammenfassung** Bedingte Kompilierung ist ein einfaches und häufig benutztes Mittel zur Implementierung von Variabilität in Softwareproduktlinien, welches aber aufgrund negativer Auswirkungen auf Codequalität und Wartbarkeit stark kritisiert wird. Wir zeigen wie Werkzeugunterstützung – Sichten, Visualisierung, kontrollierte Annotationen, Produktlinien-Typsystem – die wesentlichen Probleme beheben kann und viele Vorteile einer modularen Entwicklung emuliert. Wir bieten damit eine Alternative zur klassischen Trennung von Belangen mittels Modulen. Statt Quelltext notwendigerweise in Dateien zu separieren, erzielen wir eine „virtuelle Trennung von Belangen" durch entsprechende Werkzeugunterstüzung.

Keywords D.2.9 [Software: Software Engineering: Management]; Software product lines, variability, preprocessors, conditional compilation, views, type systems ►►► **Schlagwörter** Softwareproduktlinien, Variabilität, Präprozessoren, Bedingte Kompilierung, Sichten, Typsysteme

1 Introduction

The C preprocessor *cpp* and similar lexical tools 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, as exemplified in the code fragment below, has evolved into a common way to implement *software product lines*. Commercial product-line tools like those from *pure::systems* or *BigLever* explicitly support preprocessors. A software product line is a set of related software systems (*variants*) in a single domain, generated from a common managed code base [2]. 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 forth. Variants of a product line are distinguished in terms of *features*, 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-

[1]This summary shares text with a previous overview: C. Kästner and S. Apel. Virtual separation of concerns – A second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009.

[2]The dissertation of Dr. Christian Kästner has been awarded by the GI Disseration Award 2010. The examiners were Prof. Dr. Gunter Saake, Otto-von-Guericke-University Magdeburg, Prof. Don Batory, The University of Texas at Austin, and Prof. Dr. Krzysztof Czarnecki, University of Waterloo.

management system with transactions but without flash support.

```
 1 //Adapted code excerpt of Oracle's
       Berkeley DB
 2 static int _ _rep_queue_filedone(dbenv,
       rep, rfp)
 3   DB_ENV *dbenv;
 4   REP *rep;
 5   _ _rep_fileinfo_args *rfp; {
 6 #ifdef NO_QUEUE
 7   COMPQUIET(rep, NULL);
 8   COMPQUIET(rfp, NULL);
 9   return (_ _db_no_queue_am(dbenv));
10 #else
11   db_pgno_t first, last;
12   u_int32_t flags;
13   int empty, ret, t_ret;
14 #ifdef DIAGNOSTIC
15   DB_MSGBUF mb;
16 #endif
17   // over 100 further lines of C code
18 }
19 #endif
```

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" [11]. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability [4; 5; 11]. 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 limiting or entirely abandoning the use of preprocessors and instead implement product lines with 'modern' implementation techniques that encapsulate features in some form of modules, such as components and frameworks [2], feature modules [10], aspects [9], 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 product-line 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 intend to give a definitive answer on how to implement a product line (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

There are many arguments against lexical preprocessors, but the two most common are their lack of separation of concerns and their sensitivity to subtle errors.

Separation of concerns. 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 code of 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 product line, 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. Additionally, textual annotations can entirely obfuscate the source code and its control flow. Scattering and tangling feature code is contrary to decades of software engineering education.

Sensitivity to subtle errors. Using preprocessors can easily introduce errors at different levels that can be very difficult to detect. This already begins with simple syntax and type errors. Developers are prone to simple errors like annotating a closing bracket but not the opening one as illustrated in the code excerpt above (the opening bracket in Line 5 is closed in Line 18 only when feature *NO_QUEUE* is not 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 compilers cannot detect such errors, unless the developer (or customer) eventually builds a variant with a problematic feature combination (with *NO_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 huge number, so, even simple syntax and type 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.

3 Virtual Separation of Concerns

Instead of suggesting abandoning preprocessors in favor of more 'modular' mechanisms, as most critics do, we investigate how we can improve preprocessors. We have developed tool support that achieves a *virtual separation of concern*. Although we cannot claim to eliminate all disadvantages, we will point out some new opportunities and unique advantages that preprocessors offer.

3.1 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* [6].

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. That is, we filter irrelevant files from a file browser and we hide irrelevant code from editor windows (technically, this can be implemented like code folding in modern IDEs). In the code excerpt below, we show a view on feature *NO_QUEUE*. Note that 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 (italic and gray; e. g., in which function is this statement located) and indicate hidden code (‘▯’). Interestingly, similar context information is also present in modularized implementations in the form of extension points and interfaces.

```
static int _ _rep_queue_filedone(▯) {
#ifdef NO_QUEUE
  COMPQUIET(rep, NULL);
  COMPQUIET(rfp, NULL);
  return (_ _db_no_queue_am(dbenv));
#else
  ▯
}
#endif
```

With simple tool support for providing views, we can even emulate challenging problems of modular approaches, such as the expression problem [12] or the implementation of feature interactions [3]: Relevant source code can simply appear in multiple views.

In addition to views on individual features, (editable) views on variants are possible. 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. This goes beyond the power of modular approaches, with which the developers have to reconstruct the behavior of multiple components/plug-ins/aspects in their minds. Especially, when many fine-grained features interact, from our experience, views can be a tremendous help. Although some desirable features such as separate compilation or modular type checking cannot be achieved with views, in a similar context, Atkins et al. have measured an increase in developer productivity with views by 40% [1].

Beyond views, we also explored different visual representations of annotations that obfuscate the source code less. Among others we explored representing annotations with background colors at tool level, instead of using textual directives at source-code level. In the example below, we annotate a synchronization feature in Java code with a background color. With these fine-grained annotations at substatement level, the source code would have been highly obfuscated with traditional *#ifdef* directives. Background colors and similar visual support 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.

```
class Stack {
  void push(Object o , Transaction txn ) {
    if (o==null || txn==null ) return;
    Lock l=txn.lock(o);
    elementData[size++] = o;
    l.unlock();
    fireStackChanged();
  }
}
```

3.2 Sensitivity to Subtle Errors

Also various kinds of errors that can easily occur with *#ifdef* annotations can be detected by adding tool support. We illustrate how *disciplined annotations* can help regarding syntax errors and how new variability-aware type systems can help regarding type errors. (We did not focus on semantic errors, such as deadlocks, because they are not a specific problem of annotations but can occur equally in modular implementations.)

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 lexical preprocessor usage that considers a source file as plain text, in which every 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 entire syntactical program elements, such as classes, functions, and statements. We say an annotation is disciplined, if it aligns with the underlying structure. In our code example, the *DIAGNOSTIC* annotation is disciplined as it aligns with a local declaration, whereas the else branch of the *NO_QUEUE* annotation is undisciplined. By enforcing disciplined annotations, we can guarantee that variant generation will not introduce syntax errors in any variant.

In addition, disciplined annotations enable a clear mapping from features to code structures (instead of lexical tokens), which is beneficial for many tools, including views and type systems. 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 modular approaches, in which each module can be checked for syntax errors in isolation). Alternatively, there are tools like CIDE [6] that manage annotations at tool level and ensure that only structural elements can be annotated in the first place.

Variability-aware type systems can check that all variants in the product line are well-typed (i. e., can be compiled), without checking each variant in isolation [8]. This detects typical problems, such as functions or types that are removed in some variants but still referenced, like in the code fragment below (such problems that are less common in modular approaches since often common interfaces and separate compilation are used).

```
int[] readData() { ... }
#ifdef WRITE
int storeData(...) { ... }
#endif
int main() {
  readData();
  ...
  storeData(...);
}
```

Whereas a conventional type system checks whether it can resolve all function calls, our variability-aware type system knows about variability and checks whether all function calls can be resolved *in all variants*. 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 (the call's annotation must imply the target's annotation, a check that we can encode and solve efficiently with SAT solvers). If there is any variant in which the target but not the reference is removed (as in our example above), the type system issues an error. That is, the entire product line is checked in a single step by comparing annotations of all invocations and their respective targets, instead of checking every variant in isolation.

A variability-aware type system can efficiently check entire product lines, usually with almost constant overhead, avoiding the exponential explosion of checking variants in isolation. Type checking annotations emulates some form of modules and dependencies between them. So instead of specifying that one component imports another, we check these dependencies in scattered code using relationships between features in a product line. With disciplined annotations and variability-aware type systems, we avoid or detect the typical problems that make preprocessors so error prone.

3.3 Unique Advantages of Preprocessors

Despite all their problems, preprocessors also have unique advantages over modular approaches.

First, and most important, preprocessors have a *very simple programming model*: Developers annotate and optionally remove code. Preprocessors are very easy to use and understand. In contrast to modular approaches, no new languages, tools, or processes have to be learned. We conjecture that this simplicity drives practitioners to still use preprocessors despite all disadvantages.

Second, preprocessors are usually *language independent* and provide a *uniform experience* when annotating different artifact types. We applied our tools mostly to Java code, but explored also C code, C# code, Haskell code, grammars, documentation and other artifact types. Instead of providing a tool or language extension per language (e. g., AspectJ for Java, AspectC for C, AspectUML for UML), preprocessors (and most improvements) can be applied uniformly across languages.

Finally, annotating code does not prohibit traditional means of separation of concerns. In fact, it is reasonable to still decompose the system into modules 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 [12]. In exactly those cases, views on scattered code can be helpful for understanding. We even explored automated refactorings from annotations to modular implementations (and vice versa) [7].

3.4 Tools and Evaluation

We implemented all presented improvements in our prototype product-line tool *CIDE*, available as open source at http://fosd.net/CIDE. We evaluated the concepts and their combinations with 13 non-trivial case studies, including a product-line version of the database system Berkeley DB. We proved correctness of the variability-aware type system formally for a subset of Java.

To evaluate visualizing annotations with background colors, we conducted controlled experiments.

Currently, we explore scaling variability-aware type checking to the Linux kernel with over 10 000 features, all implemented with *#ifdef* directives and build system variability.

4 Conclusion

We have argued that preprocessors are not beyond hope for product-line development. With little tool support, we can address many problems for which preprocessors are often criticized. Views on the source code emulate modularity and separation of concerns, and disciplined annotations and variability-aware type systems detect implementation errors. Together, we name these efforts *virtual separation of concerns*, because, even though features are not physically separated into modules, a separation is emulated by tools. 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 uniformity.

We do not have a definitive answer whether physical or virtual separation of concerns is better (and this depends very much on what one measures). We are investigating different approaches in parallel and have a look at possible integration scenarios. With our work, we want to encourage researchers to overcome their prejudices (usually from experience with *cpp*) 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 product-line implementation anyway, it is well worth investing also into tool support for new preprocessors and virtual separation of concerns. *Give preprocessors a second chance!*

References

[1] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the Version Editor. In: *IEEE Trans. Softw. Eng. (TSE)*, 28(7):625–637, 2002.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, MA, 1998.

[3] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. In: *Computer Networks*, 41(1):115–141, 2003.

[4] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. In: *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[5] J.-M. Favre. Understanding-in-the-large. In: *Proc. of Int'l Workshop on Program Comprehension*, page 29, Los Alamitos, CA, 1997. IEEE Computer Society.

[6] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In: *Proc. of Int'l Conf. Softw. Eng. (ICSE)*, pages 311–320, New York, 2008. ACM Press.

[7] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In: *Proc. of Int'l Conf. Generative Programming and Component Eng. (GPCE)*, pages 157–166, New York, 2009. ACM Press.

[8] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. In: *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. accepted for publication.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In: *Proc. of Europ. Conf. Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 220–242, 1997. Springer-Verlag.

[10] C. Prehofer. Feature-oriented programming: A fresh look at objects. In: *Proc. of Europ. Conf. Object-Oriented Programming (ECOOP)*, LNCS 1241, pages 419–443, 1997. Springer-Verlag.

[11] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In: *Proc. USENIX Conf.*, pages 185–198, Berkeley, CA, 1992. USENIX Association.

[12] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In: *Proc. of Int'l Conf. Softw. Eng. (ICSE)*, pages 107–119, Los Alamitos, CA, 1999. IEEE Computer Society.

Received: October 1, 2011, accepted: October 9, 2011

Dr. Christian Kästner, Laureate of the GI Dissertation Award 2010 is a PostDoc at Prof. Klaus Ostermann's Group for Programming Languages and Software Engineering at the Philipps University Marburg, Germany. He received his Ph. D. in Computer Science for his work on virtual separation of concerns in May 2010 from the University of Magdeburg, Germany, where he was a member of Prof. Gunter Saake's Database Research Group since 2007. For his dissertation, he received the GI Dissertation Award. His research focuses on correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, and refactoring. He is the author or coauthor of over fifty peer-reviewed scientific publications.

Address: Philipps University Marburg, Hans-Meerwein-Straße, D-35032 Marburg, Germany

Preview on issue 2/2012

The topic of our next issue will be "Reactive Security" (Editor: U. Flegel) and it will contain the following articles:

- *Laskov, P. and Grozea, C.:* Anomaly detection at supersonic speed
- *Willems, C. and Freiling, F.:* Reverse Code Engineering – State of the Art and Countermeasures
- *Riviere, L. and Dietrich, S.:* Experiments with P2P botnet detection
- *Eschweiler, S. and Gerhards-Padilla, E.:* Platform-independent Recognition of Procedures in Binaries based on simple Characteristics