A Filesystem-Based Approach to Support Product Line Development with Editable Views

Diploma Thesis in Computer Sciences

by

Frank Blendinger

born 10/01/1980 in Nuremberg

Department of Computer Sciences 4
Friedrich-Alexander University Erlangen-Nuremberg

Advisors	Dipl.-Inf. Wanja Hofer Dr.-Ing. Daniel Lohmann Dipl.-Inf. Christoph Elsner Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Start of Work	04/01/2010
End of Work	10/08/2010

Abstract

Tailor-made software that can be customized with various optional or alternative features has become common both in industry and open-source environments. These software systems, often called *software product lines*, can be configured to meet varying customer needs or to fit to different environmental settings, such as the hardware systems they can run on. This has made software product lines popular for system software, such as operating systems.

A technique to implement variability is conditional compilation with preprocessors like the *C preprocessor* (CPP). By annotating code regions with directives such as *#ifdef* and *#endif*, variants of the application with different selections of these annotated regions can be generated. This simple but effective method is common for embedded software, as it leads to compact binary code and has no runtime overhead. However, the preprocessors directives and the optional or alternative code regions introduce additional complexity in the source code, which negatively impacts the maintainability of the software.

Proper tool support can help the developer to cope with the complexity of preprocessor-based variability implementations. With the help of configurable *views* on the software product line it is possible to only show the regions of the source code that belong to a selected set of features, without distracting preprocessor directives. Current approaches are all bound to specific integrated development environments. This thesis suggests to address the problem at a lower level. The presented tool, called LEVIATHAN, provides variant views in the form of *virtual filesystems*. Thereby, the use of standard tools such as syntax validators, code metric analysis programs, or arbitrary editors to view or modify a variant is made possible: These applications can work with the provided files of the virtual filesystem in the same manner as they do with regular source code files. Handling the feature annotations is decoupled from the single applications of the development toolchain, and handled transparently in the background. A major benefit and challenge is support for the automatic merging of modifications made in the virtual filesystem with the original source files.

The developed filesystem has been evaluated with the source code of the Linux kernel as an example of a large-scale software product line with over 5,000 features. The results have shown that LEVIATHAN is able to deal with the heavy usage of the C preprocessor in the Linux source tree, and is a promising approach to handle software variability in practice.

Zusammenfassung

Maßgeschneiderte Software, bei der zahlreiche optionale oder alternative Eigenschaften (*Features*) ausgewählt werden können, ist sowohl in der Industrie als auch im Open-Source-Umfeld ausgesprochen beliebt geworden. Ein derartiges Softwaresystem, das häufig als *Software-Produktlinie* bezeichnet wird, kann individuell konfiguriert werden, so dass es sich an variierende Kundenbedürfnisse oder unterschiedliche Umgebungsbedingungen, wie etwa die Hardware, auf der es laufen soll, anpassen lässt. Diese Eigenschaft hat Software-Produktlinien gerade im Umfeld der Systemprogrammierung, etwa bei Betriebssystemen populär gemacht.

Eine Technik um Variabilität zu implementieren ist die bedingte Übersetzung mit Hilfe von Präprozessoren wie dem *C-Präprozessor* (CPP). Durch Annotation von Quelltextbereichen mit Direktiven wie *#ifdef* und *#endif* lassen sich Varianten der Anwendung mit verschiedenen Kombinationen dieser Bereiche generieren. Diese einfache aber wirkungsvolle Methode ist häufig bei Software für eingebettete System zu finden, da sie zu kompaktem Binärcode und keinen zusätzlichen Laufzeiteinbußen führt. Allerdings bringen die Präprozessoranweisungen und die optionalen oder alternativen Quelltextbereiche eine erhöhte Komplexität mit sich, welche sich negativ auf die Wartbarkeit der Software auswirkt.

Geeignete Werkzeuge können den Entwickler dabei unterstützen, mit der Komplexität von Präprozessor-basierter Variabilität zurechtzukommen. Mit Hilfe von konfigurierbaren *Sichten* auf eine Software-Produktlinie ist es möglich, nur die relevanten Bereiche des Quelltextes anzuzeigen, die zu einer bestimmten Menge von Features gehören, ohne von den Präprozessor-Direktiven abgelenkt zu werden. Bisherige Ansätze sind alle an eine spezifische Entwicklungsumgebung (*Integrated Development Environment, IDE*) gebunden. In dieser Arbeit wird vorgeschlagen, das Problem auf einer niedrigeren Ebene anzugehen. Das vorgestellte Werkzeug, LEVIATHAN genannt, stellt Sichten auf Varianten in Form vom *virtuellen Dateisystemen* zur Verfügung. Dies ermöglicht den Einsatz von Standardwerkzeugen, wie Syntax-Überprüfern, Programmen zur Erstellung von Code-Metriken und beliebigen Editoren, um eine Variante zu betrachten oder zu bearbeiten. All diese Programme arbeiten innerhalb des virtuellen Dateisystems wie mit gewöhnlichen Dateien. Die Behandlung der Feature-Annotationen muss nicht mehr in jedem einzelnen Werkzeug implementiert werden, sondern kann transparent im Hintergrund ablaufen. Als herausragendes Merkmal, aber auch als größe Herausforderung wird das automatische Zurückschreiben von etwaigen Änderungen im virtuellen Dateisystem in die Originaldateien gesehen.

Das entwickelte Dateisystem wurde mit dem Quellcode des Linux-Kerns als Beispiel für eine große Software-Produktlinie mit über 5.000 Features evaluiert. Die Ergebnisse haben gezeigt, dass LEVIATHAN mit der intensiven Nutzung des C-Präprozessors im Linux-Quelltext umgehen kann, und somit einen vielversprechenden Ansatz zur Unterstützung bei der Entwicklung von Software-Produktlinien darstellt.

Contents

1 Introduction

Variability is a typical requirement for a modern software product. It may have to run on varying hardware platform or operating systems, adapt to local conditions, such as language, currencies or number formats, interact with diverse storage back ends, provide different authentication mechanisms, and so on. Sometimes an application needs to be made available in multiple editions, for example a free version, with limited functionality, and a premium version with additional capabilities. Tailor-made individual software takes this idea even further: customers can chose from a whole range of offered functionality options and have an application customized to their task-specific needs.

The implementation of variability is often realized with the help of a preprocessor. While this is a widely used technique, it leads to maintainability problems. Those, however, can be mitigated by supporting software developers with tools that make the introduced complexity of preprocessor statements manageable. In this thesis a new approach for this problem is presented: the developer is assisted a virtual filesystem that makes *views* on the source code available. These views are preconfigured variants of the software product with only those parts of the source code visible that are relevant for the specific task a developer is working on.

1.1 Variability in Software

While there are different methods to develop customizable software products, this thesis concentrates on a technique that is especially common for system software, but also found in other fields: the usage of a preprocessor tool that allows *conditional compilation*. The most widely used preprocessor for this task (and in general) is the C preprocessor (CPP). The idea is simple but effective: optional code regions are enclosed in lines like *#ifdef OPT*, *#endif*, and similar constructs. Depending on the value of *OPT*, the annotated lines will be part of the compiled variant or removed. The option values themselves are fed to the preprocessor either as command line parameters or are set in a source file, which can possibly be generated by a specialized tool responsible for the software variant configuration.

As the configuration step is performed before the actual compilation of the software, the resulting binaries can be smaller in size, consume less memory and offer better performance compared to an application that ships with every available option built in, so that computing resources are unnecessarily wasted. Smaller deployed code also has the benefit of fewer possible security risks. These advantages make conditional compilation especially attractive for the growing market of embedded systems, where heterogeneity and resource limitations are prevalent challenges.

1.2 Problem Setting

Despite its popularity and simplicity, implementation of variability with the CPP and similar preprocessors can be problematic. The main drawback is that the preprocessor directives in the source code increase its complexity and reduce the readability. This is both due to the preprocessor lines themselves, and because of the non-linear program flow they cause.

Linux kernel developer Thomas Gleixner mentioned in the keynote[1] he gave at the 22nd Euromicro Conference on Real-Time Systems [DBL10] that feature code—in this particular case, realtime extensions for the Linux kernel—implemented in terms of C preprocessor directives for conditional compilation, impedes the development process:

> `#ifdef`'s sprinkled all over the place are neither an incentive for kernel developers to delve into the code nor are they suitable for long-term maintenance.

The Linux kernel might be the most prominent software product that implements variability by means of a preprocessor, and is, with the astonishing amount of over 5,000 features [TSSPL09, SLB$^+$10], one of the largest ones, but there are many more.

Another operating system that makes heavy use of the C preprocessor is *eCos*, the *embedded Configurable operating system* [Mas02]. Figure 1.1 shows a real world example from the eCos mutex class. The class offers different strategies against priority inversion. Even though there are only four configuration variables, the code is already hard to understand. Note that the shown code is only a small section of the whole source code file, and not the only part with preprocessor statements: out of 600 non-blank lines of code, 73 are preprocessor lines.

The example clearly shows how preprocessor annotations can lead to maintenance issues when the number of features grows. The problem is not new: it has been discussed by Spencer et al. in "#ifdef considered harmful" [SC92], and more recently by Lohmann et al. in [LST$^+$06], where the situation is described as "#ifdef hell". However, preprocessor annotations are still a common technique to implement variability, and it probably will not vanish in the next few years.

[1] A transcription of the keynote, titled "The realtime preemption patch: pragmatic ignorance or a chance to collaborate?", is available at http://lwn.net/Articles/397422/.

sync/mutex.cxx

```
Cyg_Mutex::Cyg_Mutex()
{
   CYG_REPORT_FUNCTION();
   locked   = false;
   owner    = NULL;
#if defined(KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
   defined(KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
#ifdef KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
   protocol = INHERIT;
#endif
#ifdef KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
   protocol = CEILING;
   ceiling  = KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#endif
#ifdef KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
   protocol = NONE;
#endif
#else // not (DYNAMIC and DEFAULT defined)
#ifdef KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
#ifdef KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
   // if there is a default priority ceiling defined, use that to initialize the ceiling.
   ceiling  = KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#else
   // Otherwise set it to zero.
   ceiling  = 0;
#endif
#endif
#endif // DYNAMIC and DEFAULT defined
   CYG_REPORT_RETURN();
}
```

Figure 1.1: The constructor of the mutex class of the eCos operating system

1.3 Purpose and Goals

The purpose of this thesis is to design and implement a tool that helps software developers to deal with the prevalent difficulties that arise when writing or maintaining software product lines based on preprocessor annotations. Therefore, an analysis of the existing problems is to be made, which will lead to the requirements for the proposed tool.

The intended approach is the implementation of *views* on a software product line. The idea of views is to provide preconfigured versions of the variable source code that only show sections that belong to a configurable set of features. Theses views shall not simply be the output produced by a manual run of the preprocessor, but instead be created on demand for the developer, to allow interactive use.

By hiding preprocessor statements and irrelevant sections of the code, readability is regained, without losing the flexibility offered by the preprocessor annotations that are still kept in the source code. Figure 1.2 shows two views of the previous code sample

sync/mutex.cxx (Variant 1)

```
Cyg_Mutex::Cyg_Mutex()
{
    CYG_REPORT_FUNCTION();
    locked      = false;
    owner       = NULL;
    protocol    = INHERIT;
    CYG_REPORT_RETURN();
}
```

sync/mutex.cxx (Variant 2)

```
Cyg_Mutex::Cyg_Mutex()
{
    CYG_REPORT_FUNCTION();
    locked      = false;
    owner       = NULL;
    ceiling     = 0;
    CYG_REPORT_RETURN();
}
```

Figure 1.2: Views on two differently configured variants of the constructor of the eCos mutex class

from eCos. The code instantly got readable, and it is also easy to compare the two different versions. Furthermore the two function macros `CYG_REPORT_FUNCTION` and `CYG_REPORT_RETURN` are still visible in the view—by a simple run of the preprocessor, this would not be possible, as it would also replace those macros.

There are already approaches that provide such views. However, as they are all either standalone tools, or plug-ins of specific integrated development environments (IDEs), which restricts their usage in practice. The main goal of this thesis is to design and implement a tool that makes views usable for arbitrary development tools. For this purpose, the views shall be provided in the form of virtual filesystems. In addition to read-only views of a preconfigured software product line, it should also be possible to directly edit the virtual files of a view. Therefore algorithms have to be developed to merge the modifications of a preprocessed files back to the original.

Advantages and possible drawbacks of this new filesystem-based approach compared to existing IDE-based approaches shall be analyzed. Finally, the developed tool has to be evaluated for its practical usability. This includes correctness, acceptable performance, and a measurable improvement for the intended users of the tool.

1.4 Outline of This Thesis

The remainder of this thesis is organized as follows. In Chapter 2 an short introduction to software product lines in general and techniques for their implementation is given. Chapter 3 focusses on the problems that arise from the usage of preprocessors for variability implementation, followed by an overview of existing approaches to mitigate these problems. The chapter concludes with a discussion of weaknesses of current tooling, and a different approach based on virtual filesystems is suggested. In Chapter 4 a design for a tool that implements such a filesystem-based approach is introduced. First, an overview of the modular design will be given, followed by elaborated descriptions of the single components. Chapter 5 gives insight to noteworthy implementation details. The results of an evaluation of the developed filesystem is presented in Chapter 6, together with a discussion of both ways to integrate the tool into an existing development toolchain, and current limitations of the approach and how they can be solved. Chapter 7 finally summarizes the achieved goals of this thesis and gives suggestions for possible further work to extend the approach.

2 Background

This chapter gives an overview of software product lines in general, available methods for their implementation, and a brief synopsis of the current situation in academia and industry.

2.1 Software Product Lines

Traditional software engineering focusses on the development of a specific individual software system. This typically includes the phases specification, design, implementation, testing and deployment. The targeted final result is a single software product that meets all the requirements from the specification.

In contrast to this, *software product line engineering* aims at the development of multiple similar software systems from one common code base [BCK98], [PBvdL05]. A popular definition for *software product lines (SPLs)* can be found in [NC01]:

> A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The resulting software products all share a certain amount of core functionality, and are usually intended for similar purposes, but all each customized to meet the specific needs of different customers. A software product that is derived from a software product line is called a *variant*.

The parts of the software product line that can be varied are commonly called *features*. Dividing the requirements for an SPL into single features, modeling dependencies between them and related tasks are part of a process called *domain engineering*. The process of deriving a specific variant customized to the requirements of a customer from a software product line is called *application engineering*. A detailed introduction to the concepts of domain engineering and application engineering can be found in [CE00].

Compared to the development of many individual software products, software product lines benefit from the systematic reuse of code and can be produced faster, with lower costs, and at higher quality [BCK98, PBvdL05]. Software product lines are especially attractive for embedded systems, where heterogeneous hardware platforms are prevalent and resources are limited: different variants of a SPL can efficiently tailored to specific devices or use cases [BPSP04, TSH04, RS10].

2.2 Variability Implementation Techniques

While there are many different techniques to implement features, they can usually be put into two main categories: annotative and compositional approaches. In this section a brief overview of both classes is given, including benefits and drawbacks of the approaches. For a detailed comparison of annotative and compositional methods see [KAK08].

2.2.1 Annotative Approaches

Annotative approaches implement variability by annotating code regions that are intended to be features. The feature code itself is usually not separated from the common code base, and thus features are not modularized. Generation of variants is mainly done by removing annotated feature code. The most common form of such preprocessor annotations are the C preprocessor (CPP) directives *#ifdef* and *#endif*. Alternative preprocessors that provide similar facilities are, for example, *Antenna*[1] for Java ME, *GNU M4*[2], *GPP (Generic Preprocessor)*[3], or those included in the commercial product line tools *pure::variants* [BPSP04] and *Gears* [Kru07].

2.2.2 Compositional Approaches

Compositional approaches separate the feature code in distinct modules. This makes the features better manageable, as they are not scattered within the source code as it often is the case with annotative approaches. This however makes the variant generation more complex: it is not sufficient to merely remove the code fragments of inactive features from the source code base; instead the feature code has to be merged with the base code. For this purpose, different techniques exist.

A common implementation method are frameworks. A framework provides a single common platform for all variants that offers extension points, called *hot spots*, at which features can be connected [JF88]. These extension points often use design patterns like *strategy* or *observer* [GHJV95] to interact with the features.

Besides frameworks, a large amount of approaches has been developed that aim at extending programming languages to support the separation of features from a common code base. These concepts include *subject-oriented programming* [HO93], *aspect-oriented programming* [KLM$^+$97], *feature-oriented programming* [Pre97, Bat04,

[1]http://antenna.sf.net
[2]http://www.gnu.org/software/m4/
[3]http://en.nothingisreal.com/wiki/GPP

AKL09], *multi-dimensional separation of concerns* [TOHS99], *mixin layers* [SB02], and many more.

2.3 Real-Life Situation

Conditional compilation by means of a preprocessor is one of the oldest techniques to implement software variability, which might be the reason for its widespread usage. Annotative approaches have been strongly criticized in literature, both a long time before the term software product line has even been defined, for example in "#ifdef considered harmful" [SC92], and more recently by Lohmann et al. in [LST+06].

Nevertheless alternative approaches are only slowly adapted in practice, so there are still many projects that rely on a preprocessor to support variability. Especially for system software conditional compilation is common. Kästner et al. have shown that refactoring an existing software product line that relies on a preprocessor to use a compositional approach is possible [KAB07], but the process takes time and is not always practical to do. Therefore, developers will still have to cope with the difficulties of preprocessor annotations for some time.

A prominent example for a software product line based on an annotative approach is the Linux kernel: the C preprocessor has been used for its features for years, and it will probably also be used in the near future. Linux shows that it is possible to even handle large-scale software product lines with over 5,000 features [TSSPL09, SLB+10] with an annotative approach—when it is used with a certain degree of discipline.

In [EBN02] Ernst et al. have analyzed 26 software packages that use the C preprocessor (CPP). While this paper discusses many different aspects of the usage of the CPP, it also shows that the technique of conditional compilation is widely used: "conditional compilation accounts for 48 percent of CPP directives". They have also investigated the purpose of the CPP conditionals. Besides the commenting of code blocks[4] and avoidance of multiple or possibly cyclic inclusion of files by means of a construct commonly know as *include guards*[5], marking code regions as optional, or providing two or more alternative implementations, were the most common usages of the conditional statements. The authors subdivide these conditional blocks further by their purpose, for example adaptations to different hardware platforms, operating systems and libraries (portability), or the ability to change the behavior of the software, like providing messages in different languages.

So in summary it can be said that software products that implement variability with preprocessor annotative are not (yet) dying out. Still, the aforementioned maintain-

[4] `#if 0 ... #endif`

[5]
```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN
...
#endif /* FILE_FOO_SEEN */
```

ability problems exist, which makes proper tool support more important then ever. In the next chapter possible approaches to assist preprocessor-based SPL development are presented.

3 Suggested Approach

In the previous chapter it has been discussed that the usage of a preprocessor is still a common technique to implement features in software products. While this is seen as problematic by researchers, and several different approaches have been proposed, they are mainly known in academia, but not in industrial settings or the open-source community.

Most of the difficulties however can be solved or at least mitigated by proper tool support. In the following, the major issues with preprocessor-based feature implementation are analyzed.

3.1 The Need for Tools

An increasing amount of preprocessor annotations within the source code reduces its readability and therefore also its comprehensibility. This has several reasons. First, the preprocessor directives introduce clutter within the source files. They are not part of the actual program logic, so the developer has to ignore them when trying to understand the source code. On the other hand, he has to take them into consideration: depending on which conditional blocks will actually be left by the preprocessor and which will be removed, there might be a huge number of possible combinations of the annotated source code lines. The problem even increase when the blocks get larger: the lines that start or end a conditionally compiled region of code might not be visible anymore at all times in an editor; a developer then has to remember in which block the code he is currently looking at is contained. With nested blocks, things get even worse.

3.2 Existing Approaches

In this section current approaches to support the development of preprocessor-based software product lines and their limitations are discussed.[1] Tools such as CIDE [KAK08] or C-CLR [SGC07] therefore each extend a special integrated development environment (IDE) and provide preprocessed views on the configurable code base depending on a given configuration. The main disadvantage of those approaches is that they

[1]This section is based on the paper "Toolchain-Independent Variant Management with the Leviathan Filesystem" [HEB$^+$10], which the author of this thesis has co-authored.

force the developers into using that special IDE to cope with preprocessor complexity. This is infeasible both in industry projects, where toolchains are often fixed, and in open-source projects such as Linux, where the personal freedom of the developers to choose their editors and toolchains is of paramount importance[2]. Embedded software product lines, for instance, are developed in very heterogeneous setups: Engineers include domain experts in operating systems, in the actual embedded application, or specialized in drivers. Oftentimes, those engineers work in different companies supplying parts of the code, and they make use of different, special-purpose tools while developing and maintaining their subsystems, such as network analysis or real-time analysis tools. Most of those tools are either unaware of—or even incapable of dealing with—configurable source code.

Unaware tools include debuggers, for instance, which show the complete configurable base code in a debug session although only one concrete variant is being debugged at a time, possibly obfuscating program comprehension due to #ifdef cluttering. Tools that are unaware of a source code base being configurable simply do not work too well on those code bases, or they do not work to their full potential. *Incapable* tools, on the other hand, *break* when they are fed configurable source code instead of stand-alone code. Such incapable tools include many kinds of source analysis tools such as for execution time analysis, call graph extraction, deadlock detection, syntax validators, reverse engineering tools that generate UML diagrams from source code, and others. Liebig et al., for instance, report that existing tool support for Java or C# is broken by CPP conditional compilation [LAL$^+$10].

3.3 Suggested Approach

This thesis presents a new way to provide tool support for product line development based on preprocessor annotations. The currently known approaches either extend existing development products like IDEs and editors, or introduce new, additional tools. While offering a nice easement for developers, they all share a common disadvantage: they do not interact well with other tools.

Providing views on configurable software via a virtual filesystem makes them available in practically every part of the development tool chain, as they all operate on files and directories in the end. The need to re-implement handling of feature annotations in the source code in each application is eliminated. The logic is centralized in one specialized tool and provided transparently to others.

This approach fits well into the "one tool for one job" design principal, which can be seen as a certain kind of modularity, a well-known and appreciated concept in software engineering. In his book *The Art of UNIX Programming* [Ray03], Eric S. Raymond presents seventeen ideas, or rules, which are commonly known as the core of the "Unix

[2]To put it bluntly: Kernel hackers hate Eclipse.

philosophy". They provide sensible guidelines for software development in general, not only on Unix. Consider the following rules from this list:

1. Rule of Modularity: Write simple parts connected by clean interfaces.
3. Rule of Composition: Design programs to be connected to other programs.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

An IDE-based solution will always struggle to adhere to them. It is hard for other programs to interact with the feature handling component of a certain IDE. The interface of the component—if it even exists—will probably be unknown to arbitrary tools, so they have to be extended to take advantage of it. This has to be done for each external tool that needs access to the feature component. And what if the component or maybe even the whole IDE gets replaced by a different one? The new interface will probably be incompatible with the previous one, so the adaption of the external tools has to be done again.

With a filesystem-based approach however, the connectivity problem is already solved by design: any tool can handle files and directories, so providing different views to a software product line by means of virtual copies of the source code makes them available everywhere, instantly. A filesystem can be considered the most common interface to access data—it is omnipresent.

3.4 Workflow

In this section, a typical usage of the proposed filesystem, entitled Leviathan, is described.[3] Figure 3.1 shows a work flow how a target developer would use Leviathan for software maintenance. First, he localizes a given configurable code base that he wants to reason about or work on (e.g., the Linux kernel sources) in the base filesystem. Second, he defines one or more variants as sets of enabled and disabled features (e.g., #define directives). Both of those pieces of information are fed into Leviathan as input (steps 1 and 2 in Figure 3.1). The developer can then mount several variants simultaneously to different mount points by specifying the variant names (steps 3 and 4). After that, the user can operate as usual on the virtual directories and files, which are in fact slices of the original configurable code base.

[3]This section is based on the paper "Toolchain-Independent Variant Management with the Leviathan Filesystem" [HEB$^+$10], which the author of this thesis has co-authored.

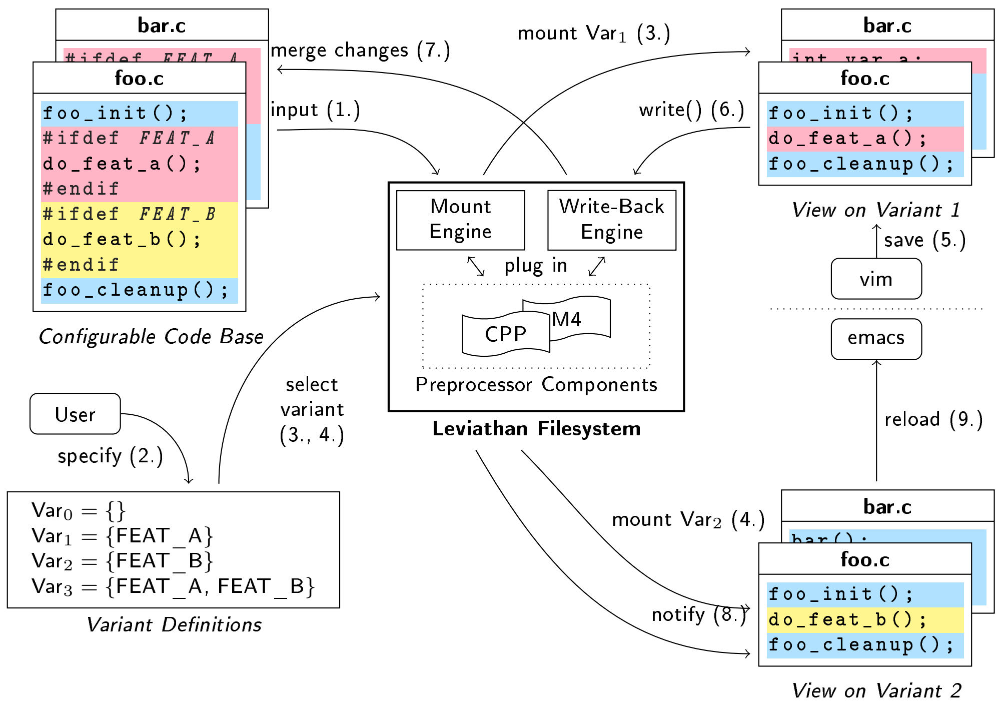


Figure 3.1: Sample workflow for SPL development using a filesystem-based approach for views

Operation includes read-only tasks such as reasoning about variants by viewing the differences between them as well as editing the virtual files with arbitrary tools; LEVIATHAN will merge back the changes into the configurable code base transparently in the background.

The work flow just described is, however, only one possible setting in which LEVIATHAN can come in handy. Four types of settings are suggested, differing in whether the actual user is *human* or a *software tool*, and whether *read-only* or also *write-back* support is necessary. Each of the following four use cases provides an example for such a usage setting:

- **WCET analysis**: A real-time analysis tool shall be used to calculate the worst case execution time of a specifically configured variant (*user is a tool, read-only access*).
- **Code reasoning**: A software developer wants to get an understanding of the source code; the code of features irrelevant for the main functionality shall be excluded to improve comprehensibility (*user is human, read-only access*).
- **Feature refactoring**: A source code refactoring tool (e.g., Coccinelle [PLMH08])

shall be applied to a certain subset of features within the code base (*user is a tool, write-back support required*).

- **Maintenance changes**: The software developer fixes localized bugs in a configured variant and wants them to be merged back to the original source code base (*user is human, write-back support required*).

The different settings result in different general requirements for LEVIATHAN. First, if a human user is involved, configurable display options help to comprehend the source code independent of the capabilities of the employed editor. In some cases, marking the beginning and end of each feature block with a dedicated marker may hinder readability, whereas, in other cases, such markers are crucial to understand the prerequisites for a piece of code to be included. Second, LEVIATHAN's write-back support must prevent or handle cases of ambiguity when merging changes back to the configurable code base. Depending on the fact whether the user is human or a software tool, one strategy or the other will be more appropriate for disambiguation. The write-back support is discussed in Section 4.7.

The decoupling from an integrated development environment has both benefits and drawbacks. As a filesystem, the views on the source code are made available via a very generic interface: system calls.This enables their usage in arbitrary tools with little effort, but it also makes the handling of the feature annotations in the source code more difficult, especially when changes are made to the virtual files of a view. The filesystem operates on a very low level and has no insight or control over specific actions performed in a tool like an editor. The developer can add, change, delete, or move lines of code. An IDE has exact knowledge of these modifications. A filesystem will only notice them indirectly, in the form of system calls of the editor process that writes the changed content of a file back to the (in this case, virtual) filesystem.

Providing views on software product lines by means of a filesystem is a new approach, and leads to different challenges then the commonly known IDE-based solutions. Chapter 4 outlines a design for a new tool that provides such a filesystem. A working prototype of the proposed design, called the LEVIATHAN *filesystem*, has been realized as part of this thesis; details about its implementation are presented in Chapter 5. A comparison of the filesystem-based approach and existing, IDE-based solutions can be found in Chapter 6.

4 Design

In this chapter the design for a tool called Leviathan is presented. Leviathan provides editable views on software product lines in the form of a virtual filesystem. Before an overview of the architecture of Leviathan is given, some terms that have a special meaning in the context of this chapter are defined.

Application. When not specified otherwise, an *application* in the following is considered to be any program that works with the files that are part of the software product line. These files can hereby both reside in the *base filesystem*, and therefore contain the source code in its original form, with feature annotations and the code of all available features, or be part of the *virtual filesystem* that provides a *view* on the software product line, with the content of the files filtered according to a given feature selection. An application may be any tool that works with files: an editor, a compiler, a validation tool, etc.

Base Filesystem. The filesystem where the source code of the software product line is located.

Base Source Tree. All source code files of the software product line including their relative structure in subdirectories. They are located in the *base filesystem.*

Feature. The term *feature* both describes a variable that is part of a *variant configuration* and the associated code regions the can be either included or hidden in a *view.*

Leviathan. The tool that implements editable *views* on software product lines. It takes the files and directories of the *base source tree* and a *variant configuration* as input, and provides preconfigured versions of the source file at a specified mount point in the form of a *virtual filesystem.*

Variant Configuration. A selection of a specific set of *features.*

View. A preconfigured version of the *base source tree.* The structure of the files and directories is the same, but the content of the source files is filtered according to a given *variant configuration.*

Virtual Filesystem. A virtual copy of the files and directories of the *base filesystem.* Virtual means that the files do not physically exist, but are created on demand by Leviathan.

With these definitions in mind, an overview of LEVIATHAN's architecture will be presented in the following. The description is held closely to Figure 4.1, that shows the main components of LEVIATHAN and how they are related with each other. The depicted connections between the single parts represent the data and information flow.

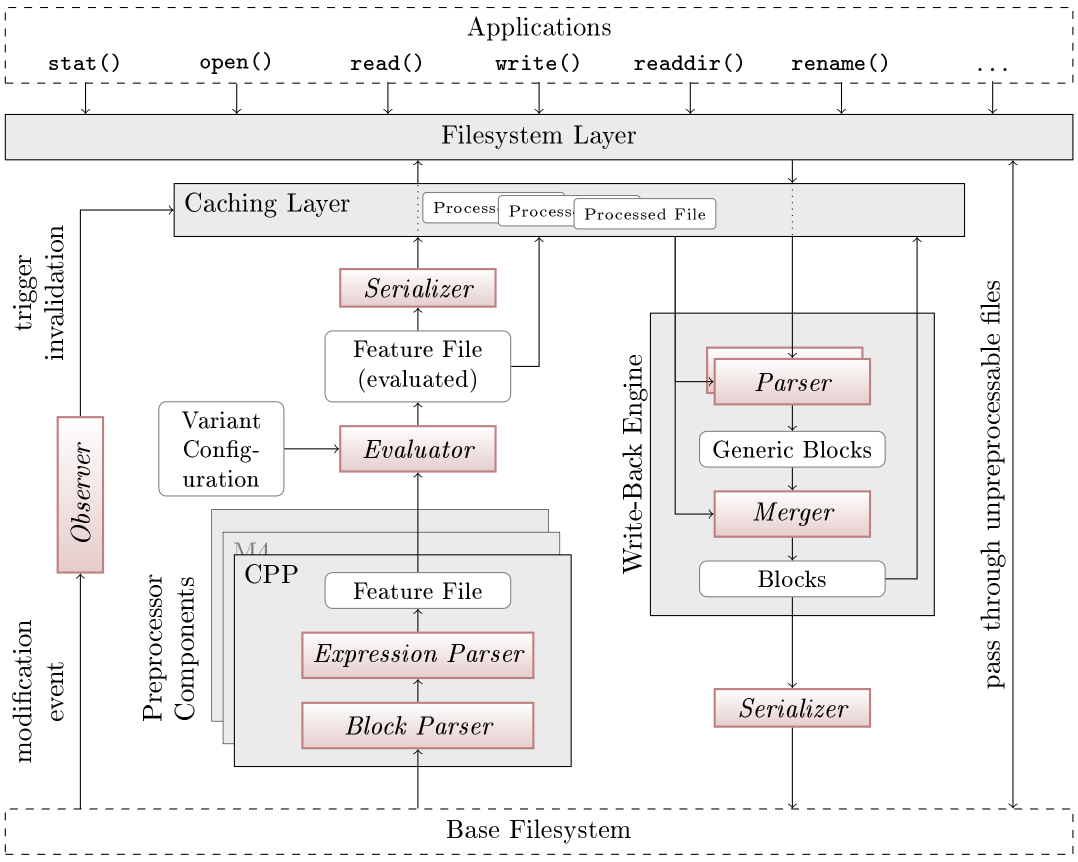


Figure 4.1: Architecture of the LEVIATHAN Filesystem

There are basically two main data flow paths in LEVIATHAN's architecture. The first one is followed when a *read* or similar request made by an application has to be handled. To fulfill the request, the respective file from the base filesystem has to be read in and parsed for its feature annotations. This is done by the *block parser* and the *expression parser*, which are part the preprocessor component. To be able to support different preprocessor languages, this component is designed to be exchangeable: the parsers are specific for a certain preprocessor language, such as CPP. The public interfaces of the component however, are generic: from the base filesystem the input is a simple character stream, while the output to the next component is a generic representation of the feature block structure found in the file, called a *feature file*.

The next part on the data flow path for a read request is the *evaluator*. This component takes both the generic feature file constructed by the preprocessor component and the variant configuration as input. The variant configuration provides values for all features of the software product line. With these values, the evaluator can decide for all blocks of the feature file whether they are part of the view or not. These evaluation results are stored in the feature files, which is the put in the cache. It is also fed to a serializer component, that will turn the structured feature file into a flat character stream consisting of the source code of only the blocks marked as *enabled* by the evaluator. The serialized version of the file is also stored in the cache, and finally used to answer the read request by the application.

The second main data flow path is followed when a *write* request has to be handled. A write operation in the virtual filesystem is actually a merge operation, as the changes have to be combined with the original version of the source file. Therefore, the parser of the write-back engine needs both the written content of the virtual file and the previously parsed original file, which is kept in the cache as input. The parser will then split the written file into block, which have to matched to original block structure of the file. Finally, the result is serialized and written back to the base filesystem. The merged blocks will be used to update the cache entry, so that they are instantly available for following read requests.

As it was already mentioned, the results of complex operations like the parsing and evaluation of annotated files are stored in a cache. This way, a big performance increase can be made for consecutive accesses to the same files. Instead of a costly recalculation, the stored results from the cache can be used to fulfill the request. Before this is done, it has to be ensured that the cache entry is still valid. This is not the case when the respective file in the base filesystem has been modified since the last access. To take care of this issue, LEVIATHAN has a base filesystem observer component, shown on the left side of Figure 4.1. This observer watches for modification events in the base filesystem and invalidates an eventual existing cache entry when they occur.

Some files need no preprocessing at all, for example binaries. For request made on those files, be it read, write, or other operations, the respective system call is just passed through to the file in the base filesystem as shown on the right side of Figure 4.1.

4.1 Filesystem Layer

The filesystem layer is the part of LEVIATHAN that is responsible for the interaction with other tools. When the filesystem has been been mounted, it provides a directory tree that is structural identical to the base source tree: every file from there will also appear under the mount point, with the same relative path. So if the developer chose for example `/usr/src/linux/` as base directory, and `/home/joe/linux-var/` as mount point, the source file `symlink.c` from the `fs/ext3/` subdirectory of the base tree can be found as expected in `/home/joe/linux-var/fs/ext3/` under the same filename. While the relative locations of the files are kept intact, their content possibly differs from the original files, depending on the configured variant—this is what makes up the view on the source code.

Interaction with other development tools, or generally any application, happens via the system call interface the operating system provides to work with files and directories. The tools can open, read and write files, get directory listings, create, rename and delete files within the mounted virtual filesystem in the same manner as they do with regular filesystems. To the applications, the implementation of the feature handling is completely abstracted. They just operate on files, and the real work is done transparently in the background.

The concrete syntax and semantics of the available system calls varies between different filesystems and operating systems. However, there is usually a common base functionality that every filesystem has to implement. This includes the opening and closing of files, listing the contents of directories, reading and writing of files, deletion and creation of files and directories, and getting and setting of metadata like ownership and permissions, time stamps, and so on. Many operating systems adhere to the IEEE Portable Operating System Interface (POSIX) standard[1], which specifies, among others, operations on files and directories via application programming interfaces (APIs) at the source level. This eases the development of portable cross-platform applications.

Most operating systems have a generic filesystem interface or an additional abstraction layer between the system calls and the filesystem drivers. Linux, for example, does the latter in form of the *Virtual Filesystem Switch (VFS)*, Microsoft Windows provides the *Installable File System (IFS)* for this purpose. Figure **??** shows the data flow between applications and the filesystem as it happens in Linux. Other UNIX- or BSD-like operating systems are very similar. All files and directories are organized within one big directory tree, independently on what kind of filesystem they reside on. When an application accesses a file, the VFS dispatches the respective system call to the filesystem the file belongs to.[2] and is transparent for the calling application. The filesystem will then perform the necessary work to fulfill the request and return the result back to the application. This is only a simplified description of the mechanisms, but it is sufficient to understand how a filesystem is connected to both the kernel and the user land applications. A more in-depth description of the VFS,

[1]POSIX:2008 or IEEE Std 1003.1-2008 represents the current version of this standard

[2]Each mounted filesystem is responsible for a subtree of the filesystem tree. Based on the complete path to a file, the VFS knows which filesystem it belongs to (not taking into consideration special cases like symbolic links).

and filesystems in general, can be found in [BC05].

The applications benefit from the transparency the VFS or similar filesystem abstraction layers offer, as they don't need knowledge of how storage handled; they can use a common interface to read from and write to files. The data might be stored locally on a hard disk drive, remotely on a different machine, accessed over a network connection, or not have a physical representation at all. The last case is usually found in *virtual filesystems*. LEVIATHAN fits into this class: the files that are presented to applications are usually not directly stored on a hard disk or other device—their content is generated on demand, depending on the preprocessor statements found in the original source code and the configuration of the view. Exceptions are files that are not preprocessed at all (for example binary files) or files with no feature annotations in them. Still, LEVIATHAN is not responsible for their physical storage. This is done by the underlying filesystem of the base source code tree, to which all requests like reading and writing are forwarded.

The LEVIATHAN filesystems is designed to be implemented on POSIX compliant systems. Therefore, its public interface is defined by the POSIX specification; functionality for a fixed set of file input and output (I/O) operations has to be implemented. The following is a list of the most important systems calls for file I/O with short descriptions for each call.

open open the file at the given path and return a file descriptor

creat create a new file at the given path and return a file descriptor

close close the given file descriptor

read read a specified amount of data from a given file descriptor into a buffer

write write data provided in a buffer to a given file descriptor

lseek reposition the current offset used for consecutive read and write operations

dup duplicate a file descriptor

link create a new hard link to an existing file

unlink delete a name for a file, and, if it was the last link to the file, the file itself

stat get file status, such as size, timestamps, ownership for a given path

access check the permissions for a file

chmod change the permissions of a file

chown change the ownership of a file

4.2 Caching Layer

The caching layer serves two main purposes: first, to reduce the processing time needed to fulfill read requests, and second, to act as a buffer for write operations that is needed to properly merge the changes with the original source files. Besides those two tasks, it also serves as a central storage unit for metadata associated with the handled files, like their sizes, access times, and similar information.

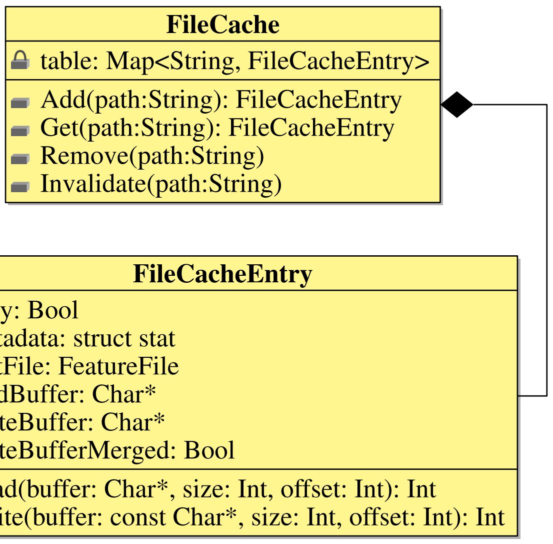

FileCache
table: Map<String, FileCacheEntry>
Add(path:String): FileCacheEntry
Get(path:String): FileCacheEntry
Remove(path:String)
Invalidate(path:String)
FileCacheEntry
dirty: Bool
metadata: struct stat
featFile: FeatureFile
readBuffer: Char*
writeBuffer: Char*
writeBufferMerged: Bool
Read(buffer: Char*, size: Int, offset: Int): Int
Write(buffer: const Char*, size: Int, offset: Int): Int

Figure 4.2: UML class diagram of the file cache

The cache can be implemented using a table like data structure. Figure 4.2 shows a simplified design for this. There will be one instance of the table class *FileCache* that can hold *FileCacheEntry* objects, each corresponding to one file of the virtual filesystem. Each entry is associated with a unique key in the table; the relative filesystem path can be used for this.

Whenever a system call with a path parameter is received by LEVIATHAN's filesystem layer that can not be served ad hoc, a cache lookup will be performed. When an entry for the requested file exists in the table, and it is marked as valid, the cached result can be used to fulfill the request. Otherwise the necessary data will be acquired and stored in a newly created cache object that is added to the cache before the result is returned.

To built a new cache entry, three other modules of LEVIATHAN are involved: the preprocessor component that will parse the feature annotations of the base source file for the entry, the expression evaluator that decides which code regions will be visible for the file in the view, and a serializer that will create the content for the read buffer. The data flow can be seen in Figure 4.1 on the left side. Both the evaluated feature file and the output of the serializer are stored in the file cache.

4.2.1 Ensuring Validity of Entries

On subsequent system calls to a virtual file that read data, the results can be directly served from the cache when a respective entry exists. This way the costly parsing and evaluation steps only have to be performed a single time for each source code file. Before the stored data from the cache can be returned, however, it is necessary to ensure that it is still valid.

A cache entry looses its validity in two cases: when there are modification made either to the virtual file the entry represents or to the underlying source file in the base filesystem. When one of these events occur at point of time after the cache entry has been created, the stored data can no longer be used. The cache entry has to be recreated from scratch before it can be used to serve another read request. As there might be an arbitrary number of additional modifications made before a new read event occurs, or there might not even be another read access, the creation of the cache object is postponed and will only when it is needed. Upon a modification, only a flag will be set to mark the cache entry as invalid, or *dirty*. This causes minimal overhead for write operations while still ensuring that only valid data will be served by the cache.

Modifications that are made on the virtual file are trivial to detect—they can only be cause by a system call a file within the virtual filesystem, which will be handled by LEVIATHAN in any case. Therefore it is enough to set the dirty flag in the respective system call handler of the filesystem layer.

To detect changes that happen in the underlying filesystem, two techniques can be used. The first is to compare a timestamp that is stored with the cache entry with the timestamp of the respective file in the base filesystem. The entry can be considered as valid, when they still match. If timestamps are not reliable, i.e. when the base filesystem does not update them when files are modified, or not available at all, a checksum has to be used. This checksum, preferably the result of a hashing function, has to be calculated when to cache entry is created, and again when the entry is accessed later. As this is obviously a more costly operation then the lookup of a timestamp, this will lead to significant performance drawbacks and should therefore only be considered when there is no alternative available.

A simpler and more reliable method is to use some kind of event notification mechanism provided by the operating system. Such a system allows applications to be notified when certain events occur, such as creation of files and directories, opening and closing of files, or modifications made to existing files. The latter type of event, modifications, is what LEVIATHAN needs to trigger an action that marks an eventually existing cache entry for the respective file as invalid. When a event-based notification system is provided by the operating system, it is preferable to make use of it; otherwise, the manual method using metadata lookups in the base filesystem has to be used.

4.2.2 Memory Consumption

For very large source trees, the cache can grow to a considerable size when a many files have been accessed. The entries are created on demand, that is whenever a file of the virtual filesystem is accessed by an application. An exception are files that are passed through from the base filesystem, such as binary files. There is no preprocessing to be done for them, so they will not be kept in LEVIATHAN's internal cache.[3] All other files will be represented in the cache by a respective FileCacheEntry object.

In the worst case, when every file of the source tree has been opened by some application, there are as many entries in the cache as there are source files. Each cache entry is made up of the parsed source code (the data structure, called a *feature file*, is described in Section 4.4.1), a serialized version of the configured source code (the original file content with only the selected feature code left and preprocessor directives used as feature annotations removed), an initially empty write buffer, and some meta data like the file size. The biggest parts of the object are the feature file and the read buffer. The feature file needs at least as much space as the parsed source file, as the whole content is stored in it. Additionally, the block structure of the feature annotations has to be embedded. The read buffer can take up any percentage between 0 and 100 of the size of the underlying source code: When all features of the file are enabled in the configured variant, or the file does not have feature annotations, it will be the same as the original file and therefore needs just as much space. However, when all code lines of the file are part of deactivated features, the read buffer will be completely empty. So as a rough estimate, a cache entry might occupy twice as much space in memory as the plain source code file.

4.2.3 Memory Management

To keep the used memory within sensible limits, the cache has to be cleaned up after some time. This problem can be solved by using a caching algorithm that will evict entries to make room for new objects [HP03]. In the current version of LEVIATHAN, no memory management has been implemented. This section merely gives an overview over possible implementation alternatives.

[3]There will probably still be some kind of caching for theses files: as they reside on another filesystem, to which the system calls are passed through, that filesystem can do caching itself. Furthermore, the operating system usually also has filesystem caches. Linux, for example, provides an *inode cache* (for files) and a *directory cache* at the VFS level, which are shared across all filesystems; additionally there is a *buffer cache* between the filesystems and underlying device drivers for hard disks, etc.

Many strategies can perform this task; the most commonly known are *Least Recently Used* (LRU) and *Least Frequently Used* (LFU) [Tan07]. There are also numerous alternative strategies to consider, which are mostly variants of either LRU or LFU, or a combination of both of them [LCK+99], [JR08]. Which one to choose highly depends on the usage scenario—none of the replacement algorithms can perform optimal in all situations. Comparisons of different strategies has been published as the results of various studies [RL96], [Zho10].

For an LRU-based cache invalidation strategy, the cache entries are ordered by the time of their last access. This can be done by using a queue, where the cache entries are inserted at the end when they are created, and moved to the end on each access. This way, old, rarely used entries end up at the front of the queue. At specific points of time, i.e. after a certain number of cache operations or when a fixed time span has passed, entries the are in the front of the queue are removed from the cache until a condition, like reducing the memory usage of the cache to i.e. 75% of the maximal allow size, is met.

The cache invalidation may only be performed for entries with a clean write buffer, i.e. files that were not opened for writing at all, or files for which the content written to the virtual file in the view has already been merged back with the feature annotated code of the original file and written to the underlying filesystem. Otherwise consecutive `write` calls from applications would lead to an inconsistent write buffer and therefore damaged files when merging the buffer content back to the original file.

Besides limiting the number of cache objects by using a replacement algorithm, keeping the entries as small as possible is essential for a small memory footprint. A good approach is to reduce the redundancy between the parsed source file and the serialized content in the read buffer. For this, a shared string pool can be used: for a continuous block of source code lines an entry is made in the pool; both the parsed file data structure and the serialized content of the read buffer can use a reference to the pool instead of actually storing the whole string themselves at the respective positions. Especially when the feature regions of a file are large, this can yield significant savings. However, there will always be a tradeoff between space (used amount of memory) and time (computation time, and therefore latency and throughput of the filesystem).

4.3 View Configuration

Each virtual filesystem that LEVIATHAN provides is associated with a configuration that specifies which parts of the source code will be visible in the preprocessed files. This configuration defines a *view*. A view is similar to a variant of a software product line, but it is used in a different context. A variant is a central idea of software product lines (see the definition given in Section 2.1), whereas a view is something a tool provides to assist SPL development. The general concept of views has been introduced in Chapter 3. In the following, the implementation of views in LEVIATHAN is explained, how it is related to SPL variants, and what the differences are.

The variant of an SPL is associated with a defined selection of features. Variants are understood as concrete configurations of the software product line, that are going to be compiled, tested, and possible shipped to a customer and intended to be deployed in a production environment in the end. Therefore, the set of feature selections has to be complete: each single feature that exists in an SPL has to be set to a valid value, that is either enabled or disabled for optional features, or one of the available values for features that provide alternatives. If the selection was incomplete, the code for the variant could not be generated and compiled.

So, for an SPL with three features, LOGGING and DEBUG, which are both optional and can be either enabled or disabled, and SCHEDULING, which is an alternative feature that can have exactly one of the three possible values *default*, *interactive*, and *real-time*, one possible variant would be defined by: LOGGING=*on*, DEBUG=*off*, SCHEDULING=*real-time*.

In addition to the need for complete configuration of features for a variant, there can also be functional dependencies between different features that limit the number of possible valid variants. Imagine an SPL that uses a database as a storage back end. It supports different databases as a feature DATABASE, with tree available alternatives: *SQHeavy*, *YourSQL*, and *Obstacle*.[4] Furthermore, there is also the optional feature TRANSACTION, that enables transaction handling to ensure data integrity. However, only *SQHeavy* and *Obstacle* support transactions, while *YourSQL* does not. Therefore, a variant with the feature selection DATABASE=*YourSQL*, TRANSACTION=*on* does not make sense and might simply not compile, or—even worse—lead to crashes in the final application when the variant was configured like this. The detection of such invalid variant configurations is part of the *domain engineering*. When dependencies between features have been defined, a variant can be verified, or the configuration process itself is done with the assistance of a tool that knows these restrictions and only allows the configuration of valid variants.

For a views, however, theses restrictions are not strictly necessary. In contrast to a variant, a view does not have to be a configuration that is meant to be built and deployed. It is merely a selection of what a developer wants to see to work on his current task as easily and efficiently as possible. Especially in combination with the aforementioned partial configurations this makes sense: to stay with the database example, a developer who is currently working on the *transaction* feature might want to see the source code of all different alternatives for the database back end, so he will probably choose TRANSACTION=*on*, DATABASE=*undecided* as configuration for his view. This is not a valid variant, but still a useful view.

[4]These are, of course, only imaginary names, and all similarities to existing databases are purely coincidental.

For large software product lines with many available features

Multiple views can be mounted in parallel at the same time, but at different mount points: each has its distinct view configuration. This can be very useful when two or more different variants of a software product line have to be compared.

4.4 Modular Preprocessor Component

The handling of feature annotations within the base source code is done by the preprocessor component. It has to perform two primary tasks. First, it parses a given input file with feature annotations in the form of preprocessor statements and provides a common data structure representing the file and its features. The actual structure, called a *feature file*, will be described in a moment. Second, in the case a view on a parsed file has been modified, the component has to be able to create the content that will be written back to the original file (after any possible merge conflicts have been solved.) This includes both the modified and the non-modified code from the view of the file, the code that was not visible in the view, and all the original preprocessor statements.

All operations that need knowledge of the used preprocessor language are centralized in this component. LEVIATHAN's other modules interact via a common API with it. The exchanged data types are either character streams for the content of the base and virtual files, or *feature files*, which are an abstraction of the feature annotations and allow a language independent implementation of the feature handling logic.

4.4.1 Feature Files

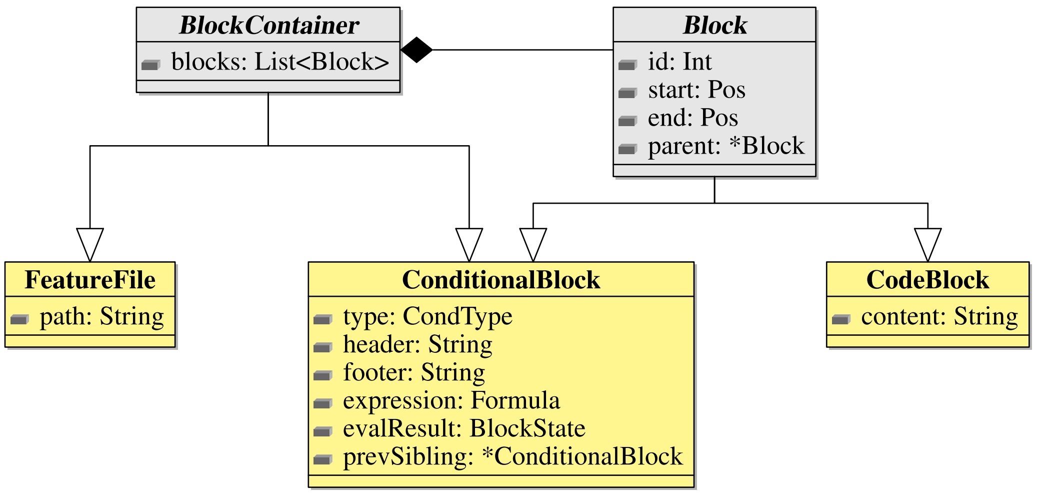


Figure 4.3: UML class diagramm of the feature file and related data structures

A *feature file* is a data structure that can store the feature regions of a source file. The file is sliced into two kind of blocks, *code blocks* and *conditional blocks*. Each block has an unique identifier, a start and an end position. The positions are stored as pairs of line numbers and columns, which allows blocks of the finest possible granularity, i.e. single characters. Most preprocessors, however, are line based, so that a block will always start on the first character of a line and end on the last character of another (or the same) line. In the following, it is assumed that blocks are always regions of complete lines. As the blocks can be nested, they are also given a level of depth, and, for convenience, a pointer to the block they are directly contained in, which shall be called the *parent* of the block.

Code blocks are merely storage units for the actual source code lines, e.g. all lines that are not feature annotations.[5] They have maximal size within a container block, so that there are never two consecutive code blocks.

Conditional blocks act as containers for other blocks. They are associated with features in the form of expressions. The expressions are transformed from the preprocessor language into an generic form, which resembles propositional logic, and can later be evaluated by the expression evaluator to configure each block for a specific variant (see 4.5.)

A conditional block can be understood as a recursive data structure—it may contain one or more other conditional blocks, which are referred to as *inner blocks*. This has implications on the features associated with the blocks: all contained blocks inherit the set of features from their outer blocks. In other terms, this means that an inner block is configured with the logical conjunction of the expressions of all blocks it is (directly or indirectly) contained in.

Figure 4.4 shows an example of the mapping from SPL files with preprocessor annotations to feature files. Depending on the active preprocessor component, the annotations are formulated in different languages. The resulting feature file is a generic representation of the structure of the underlying file, that can be used for further processing without having to take care of the preprocessor language used by the SPL.

4.4.2 Expressions

Each conditional block has an *expression* that decides in which variant configuration the block will be included. Such an expression can be a single variable that is either set to *true* or *false* in a variant, or be a formula with both logical operators such as $\wedge$ (*and*), $\vee$ (*or*), and $\neg$ (*not*), and binary relations like $\geq$, or $\neq$, and a num-

[5] Despite the fact that these blocks are called *code blocks*, their content can be arbitrary text data, as the preprocessor component will construct the feature file only based on the preprocessor statements that are feature annotations. Everything else is packed into the code blocks—this can, for example, also be documentation instead of code.

ber of variables. The formulas can be nested with parenthesis to express operator precedence.

For varying preprocessors, the syntax and availability of the operator for expressions can differ. Therefore it is the responsibility of the preprocessor component to parse these language dependent expressions, and construct formulas that will be associated with the respective conditional blocks in the feature files. The expression parser has also to take care of special preprocessor constructs such as the CPP directive *#ifdef X*, which is equivalent to *#if defined(X)*.

4.5 Expression Evaluator

As described in the previous section, each conditional block of a feature file is associated with an expression. This expression, originally formulated in the used preprocessor language, controls whether the content of a conditional block is *active* in a specific variant of the software product line, e.g. whether it will be compiled into the final program or not. In the build process of the developed SPL, this is usually done by a run of the respective preprocessor. The preprocessor will, depending on the configuration of each feature, produce a modified version of the original source code, either in memory or in the form of physical files, which is then fed to the compiler.

LEVIATHAN has to perform a similar task when it has to provide the content of file in a mounted variant of the SPL: the feature annotations have to be removed, and the content of the regions they define has to be hidden or included. This is done by iterating over the conditional blocks of the feature file data structure the preprocessor component provides, and evaluating the expressions of the blocks to *true* (*enabled*, visible in the variant) or *false* (*disabled*, hidden in the variant.)

To also support partially configured variants, a single feature may not only be set to active (*true*) or inactive (*false*), but also to *undecided.* This implies that the expression for the conditional blocks can also be evaluated to the value *undecided.* Undecided conditional blocks are included in the files of the variant, just like enabled blocks that evaluated to *true.* To give the developer the ability to distinguish those undecided blocks from the enabled blocks, an additional comment line with the expression is shown at the beginning of an undecided block.

4.6 Creating Views

Once the parsing of a base file by the preprocessor component is done, and all expressions of the resulting feature file have been evaluated, building the content for the virtual file is straightforward. It will consist of the source code lines from the underlying base file that are either not in a conditional block (and therefore not feature code), or contained in a conditional block with an expression that has been evaluated to *true* or *undecided*. The lines of blocks that were evaluated to *false* are skipped.

```
Function CreateView(FeatureFile evaluatedFile)
  content := "";
  for each Block block ∈ evaluatedFile do
    content := content + ProcessBlock(block) ;       /* iterate over the file */
  end
  return content;
```

The function `CreateView` initiates the collection of the content for the virtual file. The feature file has to be traversed in a similar manner as for the expression evaluation described in the previous section. At its top level, a feature file is a list of blocks, each of which can either be a code block or a conditional block. `CreateView` simply iterates over all these blocks and concatenates the result for each block, which is returned by `ProcessBlock`.

```
Function ProcessBlock(Block block)
  if block.type = code then
    return block.content;            /* not further nested: get all lines of code */
  else
    return ProcessConditional(block) ; /* analyze block evaluation result */
  end
```

`ProcessBlock` is given a single block, for which a type check is performed. When it is a code block, the function returns its content, the actual lines of code. When `ProcessBlock` encounters a conditional block, it calls `ProcessConditional`, that will check the evaluation result for the block, and, if necessary, descent into inner blocks.

An evaluated conditional block can have one of three results: *true*, *undecided*, and *false*. Disabled (*false*) blocks are regions of the source code that not part of the configured variant, thus `ProcessConditional` will just skip them, including any possible inner conditional blocks. Consider for example a block that belongs to the feature A with a nested block belonging to feature B. When feature A is not part of the configured variant, the whole block, including the inner lines that are annotated with feature B will not be part of the view—no matter if B is part of the variant or not.

For conditional blocks that have been evaluated to either *true* or *undecided*, the `ProcessConditional` function iterates over the inner blocks and calls `ProcessBlock` for each, just like `CreateView` has done for the outermost blocks. This way all relevant

```
Function ProcessConditional(ConditionalBlock condBlock)
content := "";
switch block.EvalResult do
    case true
        for each Block innerBlock ∈ condBlock do
            content := content + ProcessBlock(innerBlock);
        end
        break;
    case undecided
        content := content + condBlock.header;
        for each Block innerBlock ∈ condBlock do
            content := content + ProcessBlock(innerBlock);
        end
        content := content + condBlock.footer;
        break;
    case false  break;        /* skip disabled blocks including their inner blocks */
endsw
return content;
```

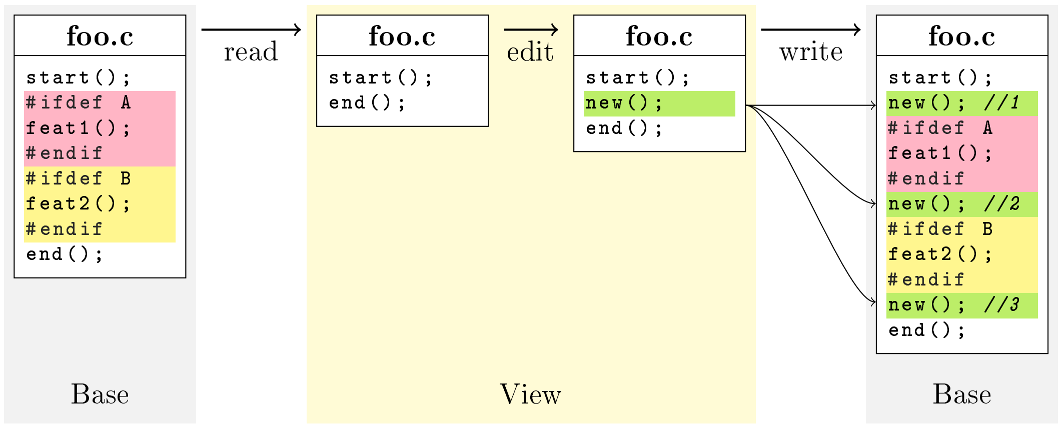
blocks of a feature file will be visited in the order of the lines of the underlying base file. The traversal resembles a classical depth-first search.

The preprocessor annotations that marked the beginning and the end of a block will be omitted for blocks with evaluation values of *true*. Only the lines of the code blocks they contain will become part of the virtual file content. For *undecided* blocks the preprocessor directives are included, as they might be valuable guidelines for the developer.

4.7 Write-Back

In addition to the readable views on variants, the provided files shall also be writable, so that the developer can edit the preprocessed source code, either manually with any text editor or IDE he likes, or automated by tools. Leviathan will merge back the changes into the configurable code base transparently in the background.

This presents a new challenge that is actually harder to solve then creating a configured, preprocessed read-only view. The configurable software base provides a clearly defined structure for each file by means of the feature annotations, i.e. preprocessor statements, within the source code. Therefore creating a view on a file is usually an unambiguous task.



```
start();
#ifdef A
feat1();
#endif
#ifdef B
feat2();
#endif
end();
```

```
start();
end();
```

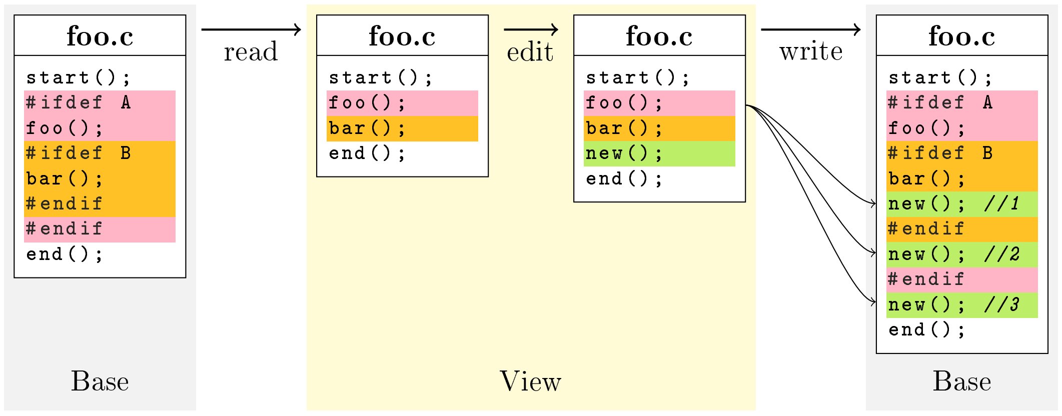
```
start();
new();
end();
```

```
start();
new(); //1
#ifdef A
feat1();
#endif
new(); //2
#ifdef B
feat2();
#endif
new(); //3
end();
```

Figure 4.5: Ambiguity problem when inserting code at the position of disabled blocks

The problem can be shown by some simple examples. Figure 4.5 shows a file with two consecutive feature blocks, both of which are disabled in a given views. The file of the view will therefore contain the code before the two blocks directly followed by the code that comes after the disabled blocks. Now the virtual file is modified: a new line is inserted between the two lines.

This leads to three possibilities for merging this change with the original source file: the first option is to place the new line before the two disabled blocks; the second is to place it between them; and the third is to put it after both blocks. While the second position is unlikely—the developer probably did not intend to introduce new code *between* two hidden code regions—the others two could both be what the developer intended. There is no way to know for sure which position is the correct one.



```
start();
#ifdef A
foo();
#ifdef B
bar();
#endif
#endif
end();
```

```
start();
foo();
bar();
end();
```

```
start();
foo();
bar();
new();
end();
```

```
start();
#ifdef A
foo();
#ifdef B
bar();
new(); //1
#endif
new(); //2
#endif
new(); //3
end();
```

Figure 4.6: Ambiguity problem when inserting code at the position of enabled blocks

In Figure 4.6 a similar example is shown: in this case, two conditional blocks are nested in each other, with A being the outer one, and B the inner one. Both feature are enable in the view, therefore all the code contained in both blocks is part of the virtual file. A new line is added by the developer to the virtual file, directly after the code that belongs to the inner block B, and directly before the code that follows the nested blocks.

Again, this leads to three possibly positions for the added code in the base file: this time, even at three different nesting levels with regard to the feature blocks. The line could be added at the end of the inner block B, at the end of the outer block A, or directly after the nested blocks.

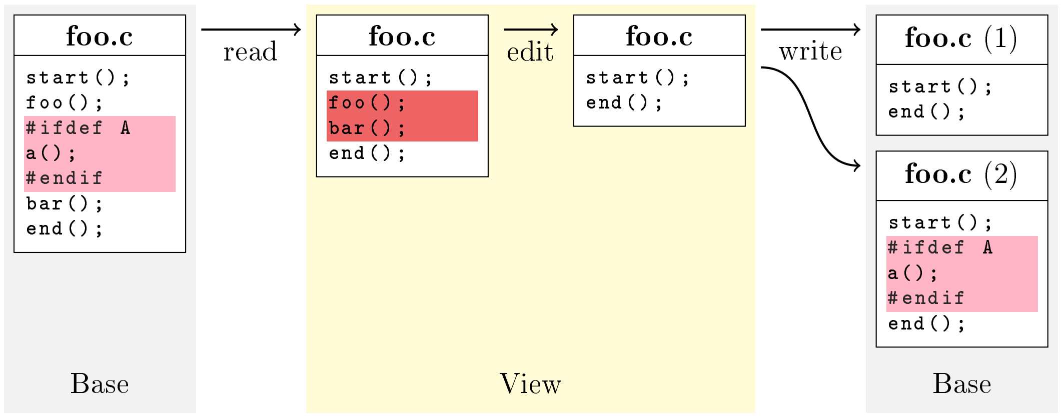


Figure 4.7: Ambiguity problem when deleting code surrounding a disabled block

The deletion of code in a virtual file can also be problematic, when the removed regions overlap feature code blocks. Figure 4.7 shows an example for such a case: the feature A is disabled in the view, therefore the code lines directly before the block and directly after the block become adjacent in the virtual file. When a region that overlaps this invisible block A is removed, it is not clear what should happen with the feature block A. It could have been indented by the developer that this block gets removed as well, as it is located within the deleted region. But then again, the code was not visible to the developer, so he might *not* want the code to be remove without him noticing.

4.7.1 Marker-Based

One solution for the block matching problem is the introduction of markers in the preprocessed source code. The markers shall be used as guidelines to rebuild the original block structure defined by the preprocessor statements that served as feature annotations in the original source file.

To make sure the matching algorithm works, the markers have to be kept in the written source file. When one or more markers are missing, or modified in a way that they can not be recognized anymore[6], an unambiguous merge can not be assured and therefore the whole write operation will fail.

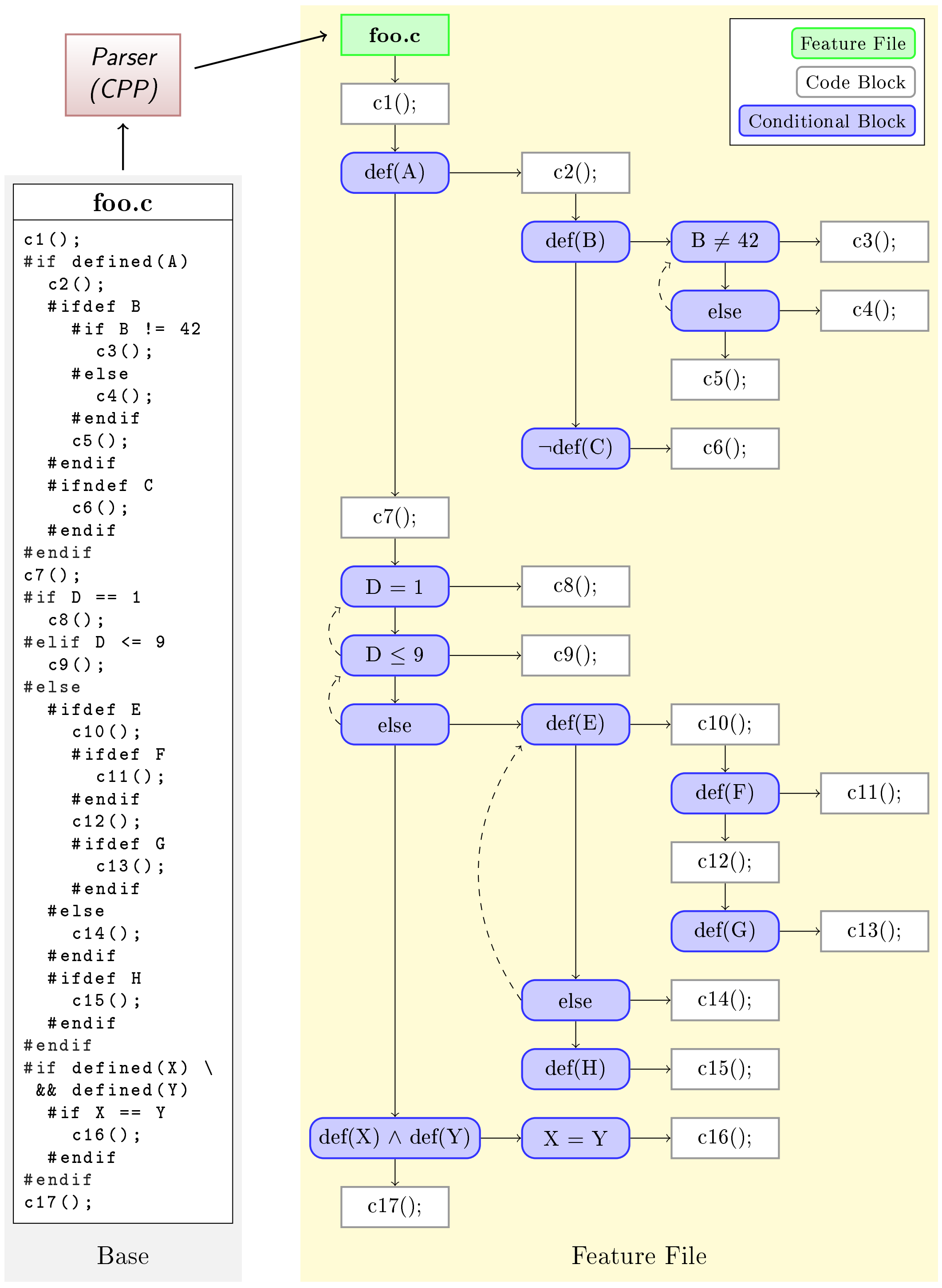
This is a behavior that, once got used to, a developer can easily deal with. It is conceptionally very similar to the mechanism that various version control systems (VCS's) use to deal with merge conflicts: when two or more different modifications of a file could not be automatically merged, the VCS shows all modifications, surrounded with markers describing their origins, together with the original content. The conflict resolution is left up to the user. In this case, the markers *have* to be removed to solve the conflict, and an error will occur when they are left in. This has proven to work quite well in practice.

4.7.2 Heuristically

In addition to the marker-based write-back strategy, different heuristics could be used to decide between the alternatives that are available when a modification cannot be merged back unambiguously. The examples in the figures 4.5–4.7 show the possible choices for some cases.

A simple method to choose would be to always insert at a specific position for the insertion of code, for example at the first position that is possible. For the deletion of regions, a similar method could be used, e.g. to never remove the code of hidden features, even when the surrounding code areas are deleted in the virtual file.

[6]for markers in the form of a single comment line, whitespace only changes to the line should not be problematic



```
foo.c
c1();
#if defined(A)
  c2();
  #ifdef B
    #if B != 42
      c3();
    #else
      c4();
    #endif
    c5();
  #endif
  #ifndef C
    c6();
  #endif
#endif
c7();
#if D == 1
  c8();
#elif D <= 9
  c9();
#else
  #ifdef E
    c10();
    #ifdef F
      c11();
    #endif
    c12();
    #ifdef G
      c13();
    #endif
  #else
    c14();
  #endif
  #ifdef H
    c15();
  #endif
#endif
#if defined(X) \
 && defined(Y)
  #if X == Y
    c16();
  #endif
#endif
c17();
```

Figure 4.4: Example of a complex feature file, built by parsing a source file with CPP annotations. The dashed arrows represent conditional blocks that belong to an *#if–#elif–#else* construct.

5 Implementation

Following the design presented in the previous chapter, a working prototype of the Leviathan filesystem has been developed. In this chapter some noteworthy implementation details will be discussed, such as the used frameworks and tools.

Leviathan depends on three external libraries: *FUSE*, on which the filesystem layer is based on, *Boost::Wave*, which is used to parse C preprocessor directives, and the *inotify* library provided by the Linux kernel, to watch the base file system for modifications.

5.1 View Configuration

The syntax of the configuration file is very similar to statements of the C preprocessor that allow the definition and undefinition of CPP macros. For optional features, that can either be enabled or disabled, the respective entries have the form *#define FEAT_X* or *#undef FEAT_X*. For features that have alternatives or can be set to a numerical value, the entries look like *#define FEAT_X VALUE*. In addition to these declaration familiar from the CPP, the keyword *#undecided* is available, to set a feature to the undecided value that has special meaning to Leviathan. Undecided features will be handled accordingly when expressions that contain those variables are evaluated, as it has been discussed in Section 4.5. An example of configuration file is shown in Figure 5.1.

spl-view-1.conf

```
#define     LOGGING
#undef      DEBUG
#undecided  DATABASE
#define     MAX_CLIENTS 16
```

Figure 5.1: Example of a view configuration file for Leviathan

The similarity to the CPP syntax makes the automatic generation of a view configuration file for Leviathan very simple, especially when the used preprocessor is the CPP. However, this is not strictly necessary. A separate tool that has domain knowledge and therefore can check feature selections for validity could be extended to create such a configuration file.

As it can be a tedious task to create a complete configuration file for a view, LEVIATHAN also provides the possibility to specify a default value for all features that are not explicitly listed in the given configuration. The default value can be any of the possible values for single feature variable, such as *enabled*, *disabled*, and *undecided*. Additionally, there is also a fourth value available for non-specified variables: *warning*. This special value will act just like *undecided*, but in addition, a warning message in the form of a comment line is put into the source code of the virtual file whereever an expression containing an unknown variable occurs. This makes it easy to spot variable missed in the view configuration file.

5.2 Filesystem Layer

Due to LEVIATHAN's design, the filesystem layer is a fundamental component. Any application using one of the provided views on a software product line communicates with LEVIATHAN in the form of system calls with paths to files or directories of the view as arguments. The filesystem layer accepts these calls, triggers the necessary actions to fulfill the request, and returns the result to the calling process. The filesystem layer can be understood as the front end of LEVIATHAN. It acts as a mediator between the "outside world", that is, applications working with files and directories, and the core components, such as the caching layer, the preprocessor, and the write-back engines.

This relationship is clear from LEVIATHAN's architecture, as described in the previous chapter—refer to Figure 4.1 for an overview. The filesystem layer is shown at the top of the figure, accepting the incoming system calls from above and passing them down. Some systems calls can be directly handled by the filesystem layer (shown on the right side of the picture). This happens in two cases.

First, for system calls where no processing has to be done at all, because there is no feature handling logic involved and the operation should therefore exactly as it would when called directly in the underlying base filesystem —those system calls are *passed through*. An example for such a operation would be `access`, which shall return the permissions for a file; it is sufficient to just call `access` on the corresponding base file and return the results. A file in the virtual filesystem is expected to have the same permission as the original file.

Second, some system calls are simple enough so that they can be handled ad hoc by the filesystem layer. This ensures good performance, as there are no additional function calls involved, and also does not introduce unnecessary clutter in LEVIATHAN's implementation.

5.3 FUSE: Filesystem in Userspace

To implement LEVIATHAN, FUSE—short for *Filesystem in Userspace*—was chosen as a basis. As the application the handles the provided filesystem can run in user space, this enables the use of additional libraries that would not be available in kernel sapce (which is typically the environment where filesystems are implemented). The architecture of FUSE is shown in Figure 5.2.

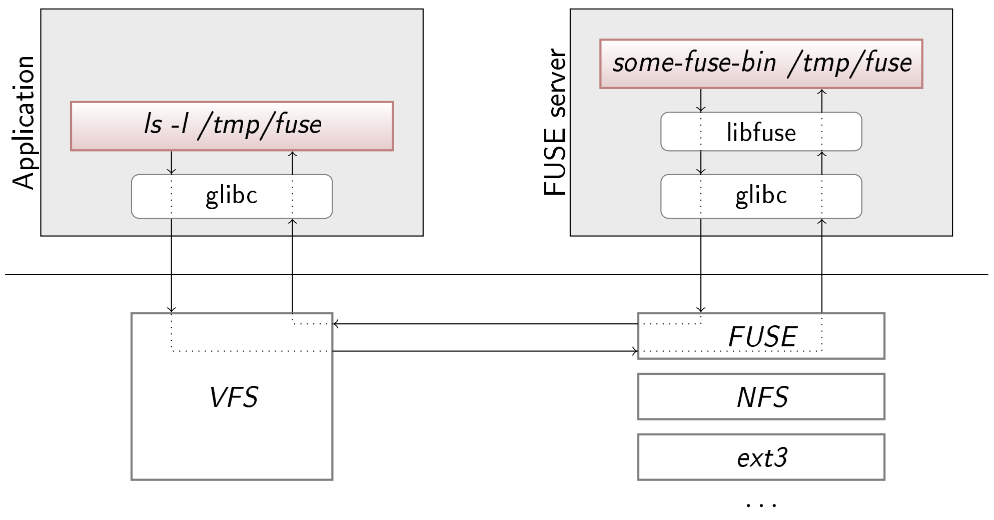


Figure 5.2: FUSE data flow

Security is also one of the main design principles of FUSE. The only operation that requires root access is the mounting, and that can be done fairly securely by a non-root user with the help of a suid program (*fusermount*).

There is also a drawback: because of the additional context switches for system calls there will always be some performance loss in comparison with a kernel driver. This might matter for filesystems where huge amounts of data have to be quickly accessed, stored, or modified. For LEVIATHAN, this is not a top priority. Its use case is providing views on the source code of software product lines. This means it has to deal with data of several hundred megabytes, or, for really large software products, maybe some gigabytes. Evaluation, as outlined in Chapter 6, suggests that the performance loss is negligible.

For all operations that are passed through as system calls to the base filesystem, the path argument is checked first. If it matches the pattern for ignored files, the operation will return an error, otherwise the respective system call will be issued with the path to the file in the base filesystem and its result is returned. This is done both for security reasons, and to be consistent with the filtered directory listings: removing

system call	description
`read`	read data from an open file
`write`	write data to an open file
`readdir`	read the content of a directory
`truncate`	change the size of a file
`getattr`	get metadata (size, timestamp, permissions, etc.) of a file, similar to stat(2) specified in POSIX[1]
`open`	open a file and return a file descriptor
`release`	close a previously opened file descriptor
`access`	check the permissions for a file
`rename`	rename a file
`create`	create a new file at the given path and return a file descriptor

Table 5.1: FUSE operations with customized implementations

system call	description
`mkdir`	create a directory
`rmdir`	remove a directory
`chmod`	change the permission bits of a file
`chown`	change the owner and group of a file
`utime`	change the access and/or modification times of a file
`symlink`	create a symbolic link
`readlink`	read the target of a symbolic link
`unlink`	remove a file

Table 5.2: FUSE operations that are passed through to the base filesystem

a file that exists in the base directory, but is not visible in the virtual filesystem is not what a user would expect and can possibly be even dangerous. This also applies for the other passed through operations like `chmod, chown`, etc.

system call	description
`statfs`	get filesystem statistics (type, free blocks, etc.); could be passed through to the base filesystem
`link`	create a hard link to a file; not needed for a virtual filesystem as LEVIATHAN
`ftruncate`	change the size of an open file; this function is optional in FUSE—when it is not implemented, `truncate` will be used instead. This is what is LEVIATHAN does.
`flush`	not implemented
`fsync`	not implemented
`setxattr`	not implemented
`getxattr`	not implemented
`listxattr`	not implemented
`removexattr`	not implemented
`opendir`	not implemented
`releasedir`	not implemented
`fsyncdir`	not implemented

Table 5.3: FUSE operations without custom implementations

The operations listed in Tableř̃eftab:fuseops-notimpl are not implemented.

5.4 Logging

Being a filesystem, LEVIATHAN has no direct communication channel to the user. It merely provides directories, files, and their contents, and handles the aforementioned operations on them. However, there are situations where the user needs additional information, especially when problems occur: a base file might not be properly preprocessed, because of syntactic or semantic errors, a write-back operation might fail because of ambiguities, and so on.

On the filesystem level, there are basically two ways to raise attention that something unexpected has happened. First, the respective system call can return an error code, and second, a message can be embedded in the file content, for example in the form of a comment.[2] LEVIATHAN makes use of both techniques. When a system

[2]This technique might be known from various version control systems (VCS's): when two or more

call cannot fulfill the request properly, it will signal this to the calling application, as it is expected by POSIX. In some situations, LEVIATHAN will embed messages into the virtual files, for example when the expression of a conditional block could not be evaluated (to make the user aware of the problem), or for guidance when the marker-based write-back is in use. This requires knowledge of the used programming language—or at least the syntax for comments in the language—as the messages may not change the semantics of source code files.

Both methods are not sufficient. Therefore, LEVIATHAN has a logging module, that provides a mechanism to report diagnostic or general messages.

different modifications of a file could not be automatically merged, the VCS shows all modifications, surrounded with markers describing their origins, together with the original content. The conflict resolution is left up to the user.

6 Evaluation and Discussion

This sections shows the results of a preliminary evaluation of the developed filesystem (Section 6.1). Furthermore ideas on how to integrate LEVIATHAN into existing toolchains will be discussed, as well as existing limitations of the approach and possible solutions (Section 6.2).[1]

6.1 Evaluation

A preliminary evaluation of the LEVIATHAN filesystem has yielded promising results. We have tested its performance by measuring the time required to read, parse, and output the complete source tree of Linux (version 2.6.31) and the eCos embedded operating system (CVS-version 2010-03-29). The test system has an Intel Core 2 Quad CPU Q9550 processor clocked at 2.83GHz, equipped with 4GB of RAM.

For Linux, the time to read, preprocess, and output (to `/dev/null`) its complete source tree of 408MB takes LEVIATHAN 130 seconds. Directly reading and outputting the source tree without employing LEVIATHAN (and therefore without preprocessing) took 14 seconds. Thus the slow down factor as caused by LEVIATHAN is about 10. As LEVIATHAN only parses the actually accessed files and we expect most use cases for LEVIATHAN to involve only a rather small number of files (a human user, for example, only can read one file at a time), we do not consider this decrease to be a show stopper. Furthermore, both the 130 and the 14 seconds were produced without caching to ensure comparable figures. When using caching (the operating system's file system caching as well as LEVIATHAN's caching), the figures decrease considerably, to 12 seconds for LEVIATHAN and to 1 second for direct reading. The fact that LEVIATHAN is still notably slower is caused by its implementation as a FUSE filesystem in user space, which by design causes expensive additional context switching overhead between kernel and user space.

When using LEVIATHAN to read, preprocess, and output the eCos embedded operating system, which has a code base of only 1MB, all figures drop well below 1 second and no noticeable disruptions occur in the work flow of a user. Although, in its current state of development, LEVIATHAN is not optimized for speed, we consider its performance sufficient for the aspired use cases.

[1]Parts of the following sections are based on the paper "Toolchain-Independent Variant Management with the Leviathan Filesystem" [HEB$^+$10], which the author of this thesis has co-authored.

6.2 Discussion

In this section it is discussed how Leviathan compares to existing tooling and how integration with those tools may be achieved. Additionally, current limitations of the approach are shown and possible solutions on how to overcome them are discussed.

6.2.1 Using and Integrating Other Tools

When a tool works on a variant file read-only (e.g., for *WCET analysis*), a separate preprocessor tool could be applied to the code base before analysis. Integrating the preprocessor with the filesystem may be more convenient than manually executing an external preprocessor, but basically both perform the same task equally well. Variability-aware *code reasoning* up to now has required dedicated viewers and editors such as CIDE [KAK08] or C-CLR [SGC07]. Our solution is generic and can be used both with the developer's favorite open source editor as well as prescribed fixed editors in industrial settings. In case of *feature-local refactorings*, some refactorings might be done with semantic patch tools such as Coccinelle [PLMH08]. However, Coccinelle detects semantic contexts based on matching normalized source code strings only. As the expressions are not evaluated, more complex Boolean conditions might be matched erroneously, resulting in patching the wrong set of code blocks. Furthermore, the patch transformations must be formulated in the Coccinelle language, whereas, with the Leviathan filesystem, arbitrary tools, such as sed, Perl, or source code transformation languages such as TXL [Cor06] may be used. *Maintenance changes* as well can be performed on a specific variant and be written back to the source code base using the developer's editor of choice.

Although Leviathan's toolchain independence allows developers to use arbitrary editors and IDEs to work on mounted variants, even in scenarios where a developer employs variability-*enabled* editors such as CIDE [KAK08] or FeatureMapper [HKW08], Leviathan may come in handy. As both have their own means for internally dealing with variability, Leviathan could be used to transparently supply them with the variability file format they require, while the actual source code variability is managed with a preprocessor such as CPP. This means that those tools can be used *complementary* to Leviathan. In that way, those tools can be seen as the independent view parts of a model–view–controller architecture; the actual preprocessing part is provided by Leviathan. For this purpose, the expression evaluator (see also Figure 4.1) would be dispensable, as these tools do not work on variant files, but on unconfigured code bases. Furthermore, to actually integrate such tools, our serialization and parsing mechanisms need to be adapted accordingly in order to be able to write and read the variability file formats of tools such as CIDE and FeatureMapper.

6.2.2 Limitations of the Approach

One current limitation of our approach is that it does not support changing the *structure* of conditional blocks in a mounted view. This means that it is not possible to add, remove, or change the inclusion condition of such a block when working on the mounted view. This limitation is unproblematic for such use cases as *feature-local refactorings* and incremental *maintenance changes* (as described in Section **??**), which do not affect the conditional structure. If, however, changes to the conditional structure are necessary, those changes can be performed directly on the configurable code base. By means of its internal notification mechanism (see also Section **??**), Leviathan will be able to update all of its mounted views where needed.

As mentioned before, Leviathan's CPP component only evaluates the subset of CPP constructs used for conditional compilation such as #if, #ifdef, and #ifndef; it leaves out #include or #define statements. As a drawback we currently cannot definitely evaluate expressions containing CPP *macros*. However, only 2 of the 27,569 conditional expressions used for feature-based configuration in Linux[2] call a macro function (e.g., `#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)` to query the kernel version). We deal with such cases by simply setting such expressions to undecided, which results in the inclusion of the corresponding block including its CPP annotations into the preprocessed file.

One very general concern about any tool that provides views on configurable code bases (such as Leviathan) is the effect of a local feature change that was performed in a view on *other features* that are not visible in the current view. Consider, for instance, renaming a variable that is also used in a hidden feature block; this refactoring will make any variant that uses that feature stop from even compiling. If such problems are to be avoided, either the write-back results can be double-checked in the configurable code base, or the change can be performed directly in the code base itself, thereby effectively avoiding Leviathan's advantage of taming #ifdef clutter. This has to be decided on a case-by-case basis, and some of the analyzed use cases (see Section **??**) will be more susceptible to that problem than others.

[2]Each preprocessor variable used for configuration starts with the prefix `CONFIG_`.

7 Summary and Outlook

This concluding chapter presents a summary of the achieved goals of this thesis (Section 7.1) and considerations about possible extensions to the implemented prototype of the LEVIATHAN filesystem (Section 7.2).[1]

7.1 Achieved Goals

This thesis shows a way to deal with the complexity of preprocessor-configured software—by using views as provided by the LEVIATHAN filesystem. The approach improves on those based on special IDEs since it enables the use of arbitrary toolchains that work directly on files. This is crucial both in industry settings with fixed toolchains as well as in open-source projects, where very heterogeneous tools and development environments are used. Although some tools may in fact be *#ifdef*-aware, LEVIATHAN *modularizes* preprocessor functionality by implementing it on the filesystem level, providing true separation of concerns.

7.2 Future Work

The approach to support development of preprocessor-based software product lines with the LEVIATHAN filesystem still needs to be fully evaluated. Especially the proposed strategies for write-back have to be implemented. The marker-based write-back method is simple enough to be added to LEVIATHAN without much difficulties; a prototypical implementation has already shown that it works. The heuristical methods, however, are currently missing a working implementation. They will have to be tested with real-world examples before they can be used in a productive environment. As explained in Section 4.7, ambiguities cannot be avoided under all circumstances. Whether the usage of a heuristic, which always brings a certain amount of possible failure with it, is feasible at all has to be shown in practice. At least as an additional mode to the safe marker-based merging, the idea is promising and should be evaluated further.

Another desirable extension is to add memory management for LEVIATHAN's internal caching layer. A concept for this has already been discussed in Section 4.2.3.

[1] Parts of the following sections are based on the paper "Toolchain-Independent Variant Management with the Leviathan Filesystem" [HEB+10], which the author of this thesis has co-authored.

Currently all entries are kept in the cache, which enables fast access to the virtual files once the have been preprocessed for the first time, but it also leads to big memory consumption. When working with big software product lines, the increased usage of resource might not be tolerable.

List of Figures

Bibliography

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[Bat04] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* O'Reilly & Associates Inc, 2005.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[Cor06] James Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[DBL10] *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010.* IEEE Computer Society, 2010.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HEB$^+$10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Toolchain-Independent Variant Management with the Leviathan Filesystem. In Christian Kästner, editor, *Proceedings of the 2nd Workshop on Feature-Oriented Software Development (FOSD 2010)*, pages 1–7, New York, NY, USA, 2010.

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping features to models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 943–944, New York, NY, USA, 2008. ACM Press.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, September 1993.

[HP03] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2003.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[JR08] Predrag R. Jelenković and Ana Radovanović. The persistent-access-caching algorithm. *Random Struct. Algorithms*, 33(2):219–251, 2008.

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 223–232. IEEE Computer Society Press, 2007.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 311–320, New York, NY, USA, 2008. ACM Press.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[Kru07] Charles W. Krueger. BigLever software Gears and the 3-tiered SPL methodology. In *Companion to the 22nd ACM SIGPLAN Conference on*

Object-Oriented Programming Systems and Applications (OOPSLA '07), pages 844–845, New York, NY, USA, 2007. ACM.

[LAL$^+$10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, New York, NY, USA, 2010. ACM Press.

[LCK$^+$99] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143. ACM Press, 1999.

[LST$^+$06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[NC01] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[PLMH08] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*, Glasgow, Scotland, March 2008.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[RL96] Benjamin Reed and Darrell D. E. Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[RS10] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*, January 2010.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[SC92] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[SGC07] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, pages 1–6, New York, NY, USA, 2007. ACM Press.

[SLB$^+$10] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '10)*, Linz, Austria, January 2010.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, third edition, 2007.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107 – 119. IEEE Computer Society Press, May 1999.

[TSH04] A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: A case of aspects in an embedded database. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04)*, Coimbra, Portugal, July 2004. IEEE Computer Society Press.

[TSSPL09] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development (FOSD '09)*, pages 81–86. ACM Press, 2009.

[Zho10] Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In Chen Ding, Zhiyuan Shao, and Ran Zheng, editors, *Network and Parallel Computing*, volume 6289 of *Lecture Notes in Computer Science*, pages 144–154. Springer Berlin / Heidelberg, 2010.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 8. Oktober 2010

Frank Blendinger