Toolchain-Independent Variant Management with the Leviathan Filesystem*

Wanja Hofer[1], Christoph Elsner[1,2], Frank Blendinger[1],
Wolfgang Schröder-Preikschat[1], Daniel Lohmann[1]

[1]Friedrich–Alexander University Erlangen–Nuremberg, Germany
[2]Siemens Corporate Research & Technologies, Erlangen, Germany

{hofer,elsner,wosch,lohmann}@cs.fau.de

ABSTRACT

Preprocessor-configured software needs tool support for the developer to be able to cope with the complexity introduced by optional and alternative code blocks in the source. Current approaches, which assist the software developer by providing preprocessed views, are all bound to a special integrated development environment. This eliminates them from being used both in industry settings (where domain-specific toolchains are often mandated) and in open-source projects (where diverse sets of editors and tools are being used and freedom of tool choice is crucial for the project success).

We therefore propose to tackle the problem at a lower level by implementing variant views at the *filesystem level*. By mounting one or more variants using our LEVIATHAN filesystem, we enable the use of standard tools such as syntax validators, code metric analysis tools, or arbitrary editors to view or modify a variant. The major benefit (and challenge) is the support for automatically writing back to the configurable code base when editing one of the mounted variant views.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Human Factors, Languages

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1.

FOSD '10 October 10, 2010, Eindhoven, The Netherlands

Keywords

Leviathan, Software Product Lines, Variability Implementation, Preprocessor-Based Configuration, Toolchain-Independent Variability Support, Filesystem Views

1. INTRODUCTION AND MOTIVATION

A lot of configurable software projects implement their variability in the sources using a preprocessor, which includes or excludes annotated code blocks depending on a given configuration. Preprocessor-based configuration is supported by all major software product line tools (e.g., pure::variants [1], Gears [7], etc.) and is especially prevalent in the domain of embedded systems and operating systems, because preprocessor-based configuration causes no run-time overhead. The matter of the fact, however, is that developers of systems such as Linux or eCos [11], an embedded configurable operating system, have to face a myriad of preprocessor directives and optional code blocks in the sources, even though they might only be working on the implementation of a single feature at a time. Thus, it has long been recognized that tool support is needed to cope with that complexity (colloquially termed #ifdef hell) to aid maintenance, evolution, and testing of such configurable software [6, 14]. For instance, in a previous study, we have found feature implementations in eCos to be highly scattered across different source files and to be tangled within the source files [9], rendering comprehension of certain files almost impossible.

Tools such as CIDE [6] or C-CLR [13] 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 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[1]. 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

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

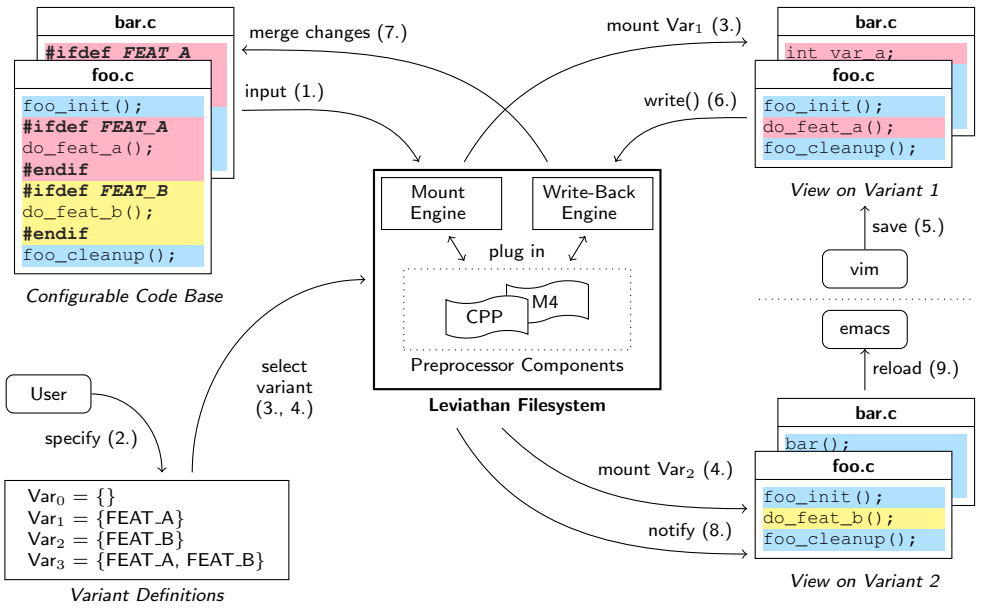

Figure 1: Example Work Flow in Leviathan.

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 [8].

In order to better support configurable software projects in industry and the open-source community, we therefore propose to implement variant views at the filesystem level. Our Leviathan filesystem is given a configurable software base with existing annotated code blocks (e.g., via CPP #ifdefs), and it can then mount views (i.e., it creates a virtual disk volume) depending on a user-provided configuration. The mounted views provide virtual directories and files, enabling the developer to use arbitrary file-based tools on that variant. Additionally, we aim at providing write-back support in Leviathan, enabling the developer to directly edit the virtual variants; the changes are then automatically merged back into the configurable code base. By providing generic, toolchain-independent, and language-independent variant views, Leviathan can therefore ideally support configurable software in industry and open-source projects.

In this paper, we first describe Leviathan in an example work flow, complemented by a classification of use cases we envision our system to be used in (see Section 2). Then, we detail Leviathan's architecture and implementation in Section 3 before providing a comprehensive description of its challenging write-back feature (see Section 4). After that, we discuss our approach and related work in Section 5 before concluding with Section 6.

Without loss of generality, for the remainder of the paper we assume the preprocessor directives to be #ifdefs and the preprocessor to be the subset of the C preprocessor CPP that is used for conditional compilation. The Leviathan architecture, however, incorporates a plug-in mechanism to support arbitrary preprocessors (see Section 3).

2. WORK FLOW AND USE CASES

Figure 1 shows an example 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 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. 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. We envisage four types of settings, differing in whether the actual user is *human* or a *software tool*, and whether *read-only*

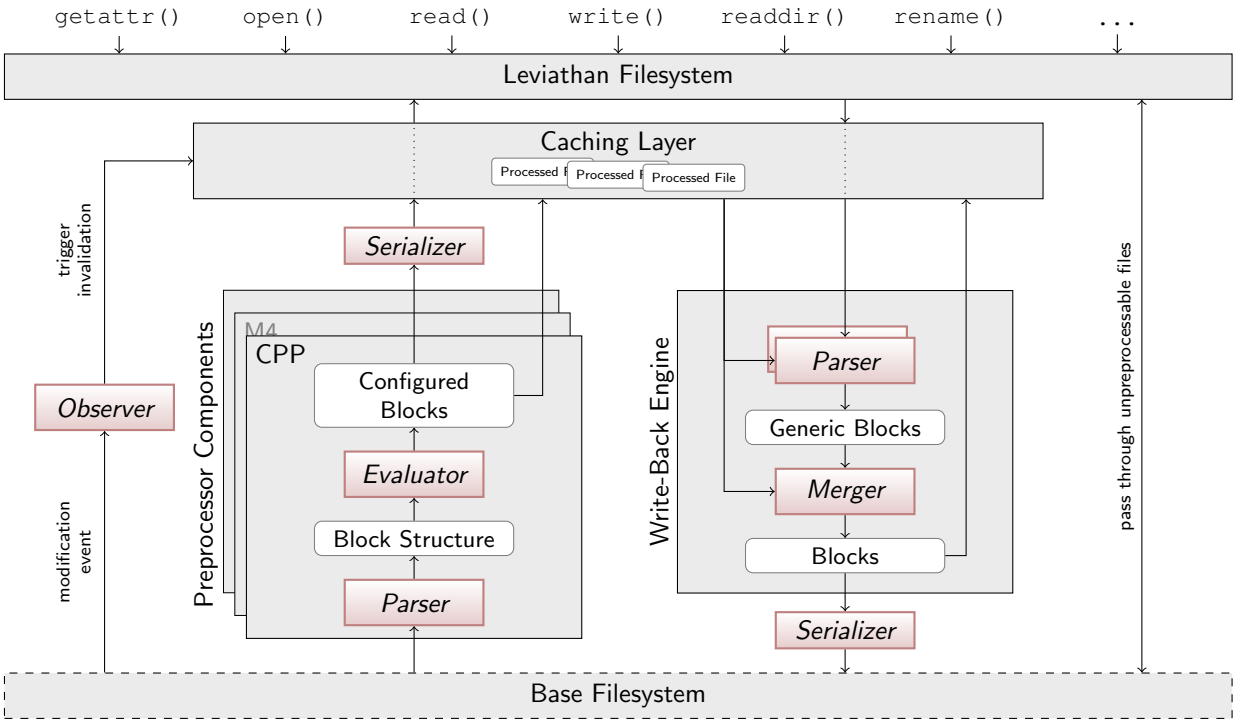

Figure 2: Leviathan Architecture and Data Flow.

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 [12]) 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. We will discuss the corresponding write-back approaches in Section 4.

3. LEVIATHAN ARCHITECTURE AND IMPLEMENTATION

In this section we will describe Leviathan's architecture, its implementation and present some preliminary evaluation results.

3.1 Architecture and Implementation

Internally, Leviathan has a modular architecture and several layers, as depicted in Figure 2.

The topmost filesystem layer communicates with applications such as editors via the standard Linux VFS filesystem interface, which includes system calls such as `open()`, `close()`, `read()`, and `write()` to be called on files. This layer is implemented as a driver for FUSE [4], a framework that allows its actual drivers to run in user space; only a very small FUSE kernel module is executed in privileged mode. Thus, the Leviathan filesystem driver can link to any third-party libraries built to support application development. FUSE supports various Unix variants as well as Mac OS X. There are also projects aiming at porting FUSE to Windows, making it the most portable framework for implementing filesystems.[2]

When processing a `read()` request, Leviathan first determines if the corresponding file needs to be preprocessed

[2]See `http://sourceforge.net/apps/mediawiki/fuse/index.php?title=OperatingSystems`.

depending on its file type. Read requests on binaries, for instance, are directly passed through to the base filesystem. If a file *does* have to be preprocessed, LEVIATHAN directs that request to its cache, which holds contents and meta data about files that have been processed before. An additional observer component monitors modification events on the base filesystem (via the `inotify` Linux kernel subsystem) and invalidates the corresponding cache entry appropriately upon changes. This is needed to synchronize both with changes made directly to the configurable code base on the base filesystem and with indirect changes to that code base via another mounted LEVIATHAN variant (see also steps 6–8 in Figure 1). Only when a cache miss occurs does LEVIATHAN direct the `read()` request to the corresponding preprocessor component.

The preprocessor component itself has a well-defined interface concerning the block structure it has to output for LEVIATHAN to work on; it effectively encapsulates the syntax parsing and expression evaluating for a concrete preprocessor, such as CPP or M4 [10]. When the content of a virtual LEVIATHAN file is requested for the first time or needs to be recalculated, the preprocessor component reads the base file from the base filesystem and uses its parsing subsystem to delineate configuration blocks encapsulated by preprocessor directives. The parser of our CPP preprocessor component, for instance, uses the Boost::Wave lexer [2] for that purpose, and it will only resolve those macros that are used for conditional compilation; that is, #defines used for definition of constants or #includes are *not* resolved in order not to impair code comprehension of the mounted variant. Each configuration block is bound to a preprocessor expression, which is stored together with the corresponding block. An expression evaluator then accesses the configuration that was handed to LEVIATHAN at mount time and uses those preprocessor variable definitions (e.g., #defines in the CPP language) to evaluate if a given block is to be included in the virtual file or not. Note that this configuration can in fact be the output of an external feature modeling tool, which assures correctness in terms of feature selection and met dependencies.

Our CPP component parses the CPP expression and evaluates them. It supports some basic arithmetical operations as the original C preprocessor does. As a special feature, logical expressions are evaluated using three-valued logic; CPP variables and whole expressions can evaluate to *true*, *false*, or *undecided*. True blocks are included in the virtual file, false blocks are excluded, and undecided blocks are output together with their preprocessor annotations. As the negation of *undecided* also evaluates to *undecided*, the #else clause of an #if/#else statement evaluating to *undecided* is also included. This way, the developer can explicitly express partial configurations by setting features to undecided, besides being able to activate and deactivate features.

The data structure as output by the preprocessor component is then stored in LEVIATHAN's cache to serve future read requests. Additionally, a serializer component computes the plain data stream that the application that has issued the `read()` call will be given as a result.

3.2 Preliminary Evaluation

Our 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 described in Section 2.

4. WRITE-BACK SUPPORT

In addition to offering read-only views for analysis tools and tasks that work on a configured variant (e.g., code metric tools or execution time analysis tools), we want to enable the software developer and his tools to *edit* a mounted variant. For instance, if he wants to perform maintenance changes (see also the corresponding use case in Section 2), this provides him with the ability to specify a configuration, mount the variant, debug it, and modify the variant code directly to get rid of the bug, eventually saving the changes in his editor (step 5 in Figure 1). In the background, the LEVIATHAN filesystem will handle the write request by the editor appropriately by directing it to its write-back engine (step 6 in Figure 1; see also the corresponding architecture part in Figure 2).

The actual write-back support is challenging. To this end, LEVIATHAN needs the additional information that the preprocessor component has computed—the mapping from *source configuration blocks* (source code blocks enclosed in preprocessor directives) to *variant blocks* (those configuration blocks actually visible in a given configuration) including their positions. Note that in the mounted view, changes in the configuration block *structure* are not supported. That is, if additional #ifdef blocks need to be introduced, for instance, this has to be done directly in the configurable code base.

Since LEVIATHAN is editor-independent, it does not have an actual edit protocol available that shows which lines were changed in which manner. Instead, it needs to operate on discrete file content snapshots provided only when the

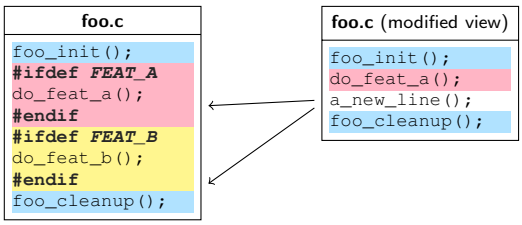

Figure 3: Example Ambiguity Problem During Write-Back.

`write()` system call is issued (by saving the changes in an editor, for instance). Such discrete snapshots make it impossible to know for certain what the user actually did and *semantically* intended to express. Consider, for instance, which block to assign a line to that was inserted exactly between two variant blocks (see Figure 3). This example is but one of several edit scenarios that causes potential ambiguity in the write-back process. We have therefore found two different ways to deal with write-back requests.

4.1 Write-Back Heuristics

Using heuristics to merge changes performed on a mounted variant view back into the configurable code base is feasible as long as two conditions hold:

1. The configuration blocks in the configurable code base are rather large and therefore there are relatively few edges between variant blocks and a lot of context for merge algorithms to work with. A recent survey covering 40 software product lines suggests that this is a valid assumption [8].

2. The changes in the mounted view are saved rather incrementally and therefore the change sets are relatively small. Again, in that case, the merge algorithms have a lot of unchanged lines serving as context to work with. Maintenance changes would be a typical use case that fulfills this property (see also Section 2).

If those assumptions hold, then LEVIATHAN's write-back algorithm will provide proper merge results to be written back to the configurable code base. If one of those conditions *does not* hold, then it is LEVIATHAN's heuristics that will make a decision in corner cases, such as which block to assign an inserted line to, as described above and in Figure 3. In any case, if the code base itself is checked into a revision control system, then the merge results can still be double-checked in a difference view before actually committing them to see whether the changes have been applied by LEVIATHAN as intended.

The actual heuristics algorithm and, with it, the question which decisions to make in critical corner cases, is currently still work in progress. There are lots of different possibilities how to match context lines that are unchanged or that changed only to a certain degree, and how to match altered variant blocks back to source configuration blocks to apply the write-back operation. In order to offer an algorithm that proves to be valuable in practice, we are preparing an analysis of typical changesets in a couple of software product lines to be able to make an informed decision about the heuristics parameters.

4.2 Marker-Based Write-Back

If the developer's changes on a mounted view have to be a hundred percent unambiguous, LEVIATHAN offers an additional, optional mechanism called markers. Markers are language-dependent comment lines that delineate former configuration blocks in the variant views. If LEVIATHAN is configured to generate marker lines at mount time, those lines are generated by the serializer component when presenting the virtual file contents to the applications (see Figure 2). By leaving the marker lines in place and only editing the lines between them, the LEVIATHAN user can fully convey what he intends to change and how.

Upon saving the changes, LEVIATHAN's write-back parsing component parses the altered virtual file for the marker lines and maps the variant blocks in between to configuration blocks in the configurable code base (see Figure 2). The merge component then updates the file block structure in LEVIATHAN's cache for future read accesses; another serializer component writes back the altered file together with all of its original preprocessor directives to the base filesystem.

With its marker mechanism, LEVIATHAN's write-back support is completely unambiguous at the cost of slightly increased clutter through the introduced marker lines. However, the mounted variant will still be a lot more maintainable than the original configurable code base, since *non-active* configuration blocks and the corresponding directives are not visible in it; only *active* blocks and their markers are displayed in the view.

Optionally, editors can implement light-weight plug-ins to interpret LEVIATHAN's markers and highlight the information in an editor-specific way, for example using vim's folds or CIDE's colors. 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.

5. DISCUSSION

In this section, we discuss how LEVIATHAN compares to existing tooling and how integration with those tools may be achieved. Additionally, we discuss current limitations of our approach and how to overcome them.

5.1 Using and Integrating Other Tools

Some of the use cases identified in Section 2 can also be addressed with existing tooling. However, our approach is the only one that is toolchain-independent and, thus, allows to work on a variant with arbitrary file-based tools. In the following, we will address each of the four use cases identified in Section 2.

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 [6] or C-CLR [13]. 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 [12]. 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 [3] 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 [6] or FeatureMapper [5], 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 2) 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.

5.2 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 2), 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 3), 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[3] 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 2) will be more susceptible to that problem than others.

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

6. CONCLUSION AND FUTURE WORK

We have shown a way to deal with the complexity of preprocessor-configured software—by using views as provided by our LEVIATHAN filesystem. Our 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.

We still need to fully evaluate our LEVIATHAN approach and especially its write-back engine and approaches to be able to exactly state its benefits and disadvantages; the evaluation targets will be Linux and eCos, as well as a department-internal operating system product line that is used in classes. In future work, we additionally want to tackle read and write support for code bases that implement optional features with patch sets, which are, for example, prevalent for freshly implemented and experimental features in Linux. Furthermore, we are working on a formalization of our two write-back approaches to be able to grasp their respective advantages and boundaries—and, therefore, their applicability to different use cases.

7. REFERENCES

[1] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf%`, visited 2010-08-23.

[2] Wave V2.0: Wave C++ preprocessor library. `http://www.boost.org/doc/libs/1_43_0/libs/wave/index.html`, visited 2010-07-29.

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

[4] FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`, visited 2010-07-29.

[5] F. Heidenreich, J. Kopcsek, and C. 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.

[6] C. Kästner, S. Apel, and M. 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.

[7] C. 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.

[8] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. 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.

[9] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. 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, Apr. 2006. ACM Press.

[10] GNU M4 – GNU Project – Free Software Foundation (FSF). `http://www.gnu.org/software/m4/`, visited 2010-07-29.

[11] A. Massa. *Embedded Software Development with eCos*. New Riders, 2002.

[12] Y. Padioleau, J. L. Lawall, G. Muller, and R. R. 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, Mar. 2008.

[13] N. Singh, C. Gibbs, and Y. 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.

[14] H. Spencer and G. 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.