Aspect mining in the presence of the C preprocessor

Bram Adams
Herman Tromp
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium
{bram.adams,herman.tromp}@ugent.be

Bart Van Rompaey
LORE, University of Antwerp
Middelheimlaan 1
B-2020 Antwerpen, Belgium
bart.vanrompaey2@ua.ac.be

Celina Gibbs
Yvonne Coady
MOD Squad
University of Victoria
Canada
{celinag,ycoady}@cs.uvic.ca

ABSTRACT

In systems software, the C preprocessor is heavily used to manage variability and improve efficiency. It is the primary tool to model crosscutting concerns in a very fine-grained way, but leads to extremely tangled and scattered preprocessor code. In this paper, we explore the process of aspect mining and extraction in the context of preprocessor-driven systems. Our aim is to identify both opportunities (extracting conditional compilation into advice) and pitfalls (mining on unpreprocessed code) in migrating preprocessor code to aspects. We distill five trade-offs which give a first impression about the usefulness of replacing the preprocessor by aspects. Preprocessor-driven systems prove to be a real challenge for aspect mining, but they could become on the other hand one of the most promising applications of AOP.

1. INTRODUCTION

Systems software manages and controls the hardware to support the tasks of application software. Over the years, a body of large, long-living systems has accumulated, entailing operating systems, device drivers, compilers, virtual machines, etc. Many of these systems have configurable parameters built into the source code to tailor the product to one specific hardware platform or to allow different variants to be composed. Usually, such systems are developed in C/C++, with the C preprocessor handling this variability. Typically, these preprocessor constructs are cross-cutting concerns, scattered and/or tangled in the program.

Most current aspect mining approaches are geared towards modern OO languages like Java or Smalltalk. Bruntink et al. [5, 4], however, deal with a large C code base in which five known programming idioms for error handling, logging, etc. are prime candidates for refactoring into aspects. Those idioms involve the use of standard function-like macros (only one of the three preprocessor constructs), but it turns out that automatic aspect refactoring of macro expansion is severely hampered by inconsistent application of the macros [4]. Bruntink et al. have found that the huge number of extracted advice variants combined with the high need for advice context diminishes AOP's benefits in localising crosscutting behaviour in this case. We are interested to find out whether these findings can be generalised to preprocessor usage in general.

After summarising the three important preprocessor constructs (Section 2), this paper looks in more detail at aspect mining and refactoring in preprocessor-driven source code (Section 3). We consider this topic in an exploratory way to highlight challenges ahead and to consider currently existing solutions for subproblems. More in particular, in Section 4 we illustrate how traditional aspect mining techniques which operate on preprocessed code, and preprocessor-aware mining techniques relate to each other. From our exploration, we distill five trade-offs which give a first indication about the feasibility of using aspects instead of the preprocessor (Section 5). Finally, Section 6 presents our conclusions.

LATE '08 Brussels, Belgium

2. THE C PREPROCESSOR

The `cpp` is a simple, but powerful tool which lives in a symbiotic relation with the C compiler. Many shortcomings of the C language can be resolved with preprocessor directives. However, its great flexibility and the fact that it does not respect the C syntax rules make `cpp` a potential source of suboptimal coding practices and may cause confusion [19]. `cpp` is a necessary evil for every C programmer and maintainer. Even the advent of C++, which solves many shortcomings of C which previously had to be handled with preprocessor directives, did not eradicate the use of `cpp` from the programmer's tool box [16].

From the program understanding view point, `cpp` complicates the analysis of source code considerably. The programmer actually writes two programs: the `cpp` program and the C program. The result of the `cpp` program can be one of a multitude of C programs. Someone wanting to parse a system for program understanding purposes has two options: either choose one particular configuration and parse the preprocessed code for that configuration, or make the parser more robust so that it understands the unpreprocessed code, which does not have to be valid C.

In particular, the `cpp` introduces the following constructs:

`#define` Macro definition and expansion. `cpp` understands the directives `define` and `undef` for the definition of macros. Macros are commonly distinguished based on whether they require parameters (*function-like*) or not (*object-like*).

`#include` Copy mechanism traditionally used to emulate a module/interface system. The referenced files are pasted literally inline. File inclusion usually introduces declarations of functions or types into a source file.

`#ifdef` Conditional compilation is used for parametrisation of source code based on the build-time configuration.

Standalone analysis of the C preprocessor has been well-studied. Ernst et al. [8] have measured that on average 8.4% of lines of source code contains preprocessor directives. Conditional compilation takes up 48% of these, macro definitions 32% and file in-

```
#define SWAP_BYTES(a) \
    ((((a) << 8) & 0xff00) | 0x00ff) \
    & (((a) >> 8) | 0xff00)
```

Figure 1: Macro definition in `zip.cpp`.

clusion 15%. 25% of the source code contains macro expansions while 37% is controlled by conditional compilation logic. Conditional compilation is especially popular for managing portability of an application (37% of its usage patterns) and for avoiding accidental redefinition of a preprocessor flag (17%).

In addition to heavy usage, preprocessor-driven source code is plagued by understandability problems caused by complicated macro bodies, side-effects and dynamic scoping, unsafe and inconsistent usage, multiple definitions, etc. The fine-grainedness of conditional compilation at the same time is its biggest enemy, as source code rapidly becomes a maze in which normal program flow is hard to distinguish from (often nested) conditional compilation checks. Often, similar conditionally compiled code segments can be seen scattered throughout the system. Singh et al. claim that for these reasons conditional configuration code could be extracted into aspects [17]. Based on case studies with their configuration browser (C-CLR), they state that "The problems associated with conditionally compiled configurations include redefined macros, non-reusable configuration code, and non-explicit representation of configuration compositions". This paper explores the process of automatic mining and extraction of aspects from preprocessor code.

3. ASPECT MINING IN PRESENCE OF PREPROCESSOR

In this paper, we make a couple of assumptions. First, we presume that the intended goal is to replace preprocessing constructs as much as possible by aspects. As an alternative, variability could be made a responsibility of the build system or be delayed until run-time, but we do not consider this.

Furthermore, we do not consider "undisciplined" or "dirty" preprocessor usage, i.e. preprocessor usage which defies C's scoping or syntax rules. This is in line e.g. with Garrido et al.'s work [9]. They only allow preprocessor directives on statements, declarations, structure fields, enumerator values and array initialiser values. Ernst et al. [8] point out that roughly two thirds of preprocessor usage corresponds to such simple patterns. This excludes amongst others conditional compilation of parts of tokens or macros which contain incomplete syntactic structure. For such undisciplined preprocessor usage, Baxter et al. [3] even claim that "The reaction of most staff to this kind of trick is first, horror, and then second, to insist on removing the trick from the source".

With these assumptions in mind, we can consider the implications of using AOP on each preprocessor directive.

Macros. Many macros behave like inline function definitions. The `SWAP_BYTES` macro (Figure 1) of the HotSpot VM[1], e.g., swaps the bytes[2] of the variable passed on as parameter `a`. This functionality is used throughout the VM just like an ordinary function, but it has been expressed as a macro for performance reasons.

If existing aspect mining techniques like fan-in [14] or clone detection [5] would accept unpreprocessed source code, occurrences of (unexpanded) macro expansions could become (part of) candidate seeds and possibly end up inside the extracted advice body. As long as the candidate seeds correspond to valid C code segments, they may even contain expansions of undisciplined macros themselves. The only caveat is that the extraction of advice from these seeds will be more complex, as all involved macro definitions, free variables, etc. will have to be extracted too.

[1] All our examples are drawn from this VM implementation.

[2] This behaviour is only defined if `a` has an integer type.

```
#ifdef LOG_THREADS
    dprintf(2, "[Suspending fromtid = %ld,
        tid = %ld, pid = %d, count = %d]\n",
            pthread_self(), tid->sys_thread,
            tid->lwp_id, count);
#endif
```

Figure 2: Conditional compilation (C1) in `threads_linux.c`.

```
JNIEXPORT jlong JNICALL
Java_sun_awt_X11GraphicsDevice_getDisplay(
 JNIEnv *env, jobject this){
#ifdef HEADLESS
    return NULL;
#else
    return (jlong) awt_display;
#endif /* !HEADLESS */
}
```

Figure 3: Conditional compilation (C2) in `awt_GraphicsEnv.c`.

File Inclusion. File inclusion is special, as opportunities for extracting inclusion into advice are rare. An example would be the case where each file needs to include a common header file with debugging macros. A dedicated advice construct for file inclusion would be needed in this (rare) case. The advice construct as we know it cannot be used to include a header file, as included files may contain type or function declarations.

Conditional compilation. Conditional compilation is the most interesting construct, as it causes most program understanding problems. It enables build-specific variants of the source code, i.e. preprocessed code corresponds to one particular build configuration. Aspect mining should be performed on unpreprocessed source code, because otherwise the identified aspects would not be applicable in all build environments.

To simplify the discussion, we have distilled five important categories of conditional compilation usage in the HotSpot VM:

C1 A recurring conditional compilation block which is scattered around the code base just to add some extra code. A typical example is logging, as exemplified in Figure 2. This particular logging block occurs nine times in the HotSpot VM.

C2 A recurring conditional compilation block which is scattered around the code base to replace some code. The example of Figure 3 shows a compilation block which manages the Java windowing libraries in a headless environment.

C3 A recurring conditional compilation blocks extending existing conditional branch structures. In Figure 4, we notice how an additional case statement is added (at multiple places).

C4 A preprocessor #if-#elif-#else block is a variation point for a changing target platform, e.g. the operating system. Each block represents an alternative implementation. In Figure 5, every conditional block encapsulates a function.

C5 Conditional blocks can be nested freely. As a complex example, consider the multitude of combinations in `awt.h` because of nested conditional compilation (Figure 6).

It is important to note that an aspect candidate may be involved in multiple categories of interactions, e.g. because of nested conditional compilation. Some cases like C1 and C5 are clearly aspect candidates. Modularising the conditional code will clearly im-

```
#if !defined KCMS_NO_CRC
 case 'c':
        fd->fd.crc32 = 0xFFFFFFFFL;
        fd->type = KCMS_IO_CALCCRC;
        break;
#endif
```

Figure 4: Conditional compilation (C3) in `awt_GraphicsEnv.c`.

```
#if defined (KPWIN16)
 void FAR KpSleep (KpUInt32_t ms,KpInt32_t d){
   /*Windows 16 bit implementation*/
 }
#elif defined (KPWIN32)
 void FAR KpSleep (KpUInt32_t ms,KpInt32_t d){
   /*Windows 32 bit implementation*/
 }
...
#else
 void FAR KpSleep (KpUInt32_t ms,KpInt32_t d){
   /*generic implementation*/
 }
#endif
```

Figure 5: Conditional compilation (C4) in `sync.c`.

prove program understanding. An alternative for C1 could be to extract the code into a function and call it instead. At first sight, C2 does not seem to be a straightforward aspect candidate. However, it is easy to imagine around-advice which nullifies the return value of the procedures in which the conditional logic resides. C4 could be tackled by delegating the conditional logic to the build system, i.e. by moving all procedures of a given platform into a dedicated file and by making the build decide which one to compile. However, this may lead to an exponential number of files because of the arbitrary nesting of configuration checks, and because it delegates important knowledge of configuration constraints to the build system, which faces serious understandability problems on its own [1]. Alternatively, one could extract different procedure definitions into separate around-advices, of which the pointcut depends on the build system configuration. Finally, C3 is the most complicated example, as it requires fine-grained join points on conditional branches. This is a good example of the fine-grained nature of typical preprocessor usage. The overall challenge of aspect mining in preprocessor-driven code is to take into account all possible builds which can emerge from the unprocessed code.

4. MINING AND REFACTORING PROCESS

Now that we have identified the possible applications of AOP on the three preprocessor constructs, we take a look at the general process of aspect mining and refactoring in preprocessor-driven source code. We do this chronologically, starting with mining activities and then considering refactoring.

4.1 Aspect mining

Detection of seeds Standard mining approaches [11] can be used to discover candidate seeds in preprocessor-driven code, but they should be amended to work on unpreprocessed code. This is not as straightforward as it sounds, because all three preprocessor constructs can abstract over irregular programming constructs, like incomplete function definitions, partial struct declarations, etc. Hence, parsers need to be robust with reference to the program's structure. Vidács and Beszédes [21] describe a schema for storing parsed preprocessor information that enables to extract the original source

```
#ifdef NETSCAPE
#else  /* ! NETSCAPE */
 #ifdef DEBUG_AWT_LOCK
  #ifndef XAWT
   #ifdef JNI_AWT_LOCK
   #else /* ! JNI_AWT_LOCK */
   #endif /* ! JNI_AWT_LOCK */
  #else /* XAWT */
  #endif
 #else /* ! DEBUG_AWT_LOCK */
  #ifdef JNI_AWT_LOCK
  #else /* ! JNI_AWT_LOCK */
  #endif /* ! JNI_AWT_LOCK */
 #endif /* ! DEBUG_AWT_LOCK */
#endif /* ! NETSCAPE */
```

Figure 6: Conditional compilation directives (C5) in `awt.h`.

code from such a model again. Spinellis' CScout [20] uses the concept of "token equivalence classes" to link back the occurrences of identifiers in preprocessed code to the unpreprocessed version. Somé [18] uses the naive approach of parsing all possible paths, but is able to speed things up with some heuristics. Finally, McCloskey et al. [15] have proposed an AST-based replacement macro language called "ASTEC".

Fan-in analysis would be useful for detecting scattered macro expansions, as those can be interpreted as normal function calls. As the `SWAP_BYTES` function-like macro of Figure 1 is expanded 51 times by three implementation files, these usage sites can be considered an aspect candidate based upon the fan-in principles.

Conditional compilation regions on the other hand could be identified more easily by e.g. clone detection tools. CCFinder[3] [10] is a clone detection tool which is able to work on unpreprocessed code. The C1 and C2 examples on Figure 2 and 3 respectively could be identified like this, because they occur in many places with only minor variations. C4 (Figure 5) is different, as the various platform-specific code pieces deviate quite a bit from each other. This means that the duplication detection tool needs to be robust with reference to local modifications.

Any conditional region which is not flagged as scattering, should still be considered as a possible tangled seed. Human intervention needs to decide whether it really gives rise to tangling or not.

Categorisation of seeds. Once interesting candidate seeds have been found, an extra (automated) step is required to find out whether candidate seeds involve conditional compilation directives or not. If they are, this means that the seed is highly dependent on build-time configuration info. The role of this info inside the seed can vary, and this can influence the difficulty of expanding the seed and of extracting useful advice from it.

Fortunately, categorising the seeds can be done automatically. It involves checking whether seeds cross conditional compilation region boundaries or not. Based on our analysis in Section 3, there are only three categories of seeds: *external* (Figure 7a), *internal* (Figure 7b) and *cross* (Figure 7c).

"External" seeds do not contain any conditional compilation segment, similar to the situation traditional aspect mining approaches encounter. The seeds may refer to macros or file inclusion, in which case these will end up inside the extracted advice. The extracted aspect will have to be processed by the preprocessor before the weaver can do its work. This is no distractor, as it is especially conditional compilation which threatens program understanding.

At the other extreme, "internal" seeds are located completely inside conditional compilation segments, cf. C1 up to C4. These

[3] `http://www.ccfinder.net/`

```
1  int i=666;
2  .../* no conditional compilation */
3  int a=f(i,"bla");
```
(a)

```
4  #ifdef LINUX
5  int i=666;
6  .../* no constraints */
7  int a=f(i,"bla");
8  #endif
```
(b)

```
9  int i=666;
10 #ifdef LINUX
11 .../* no constraints */
12 int a=f(i,"bla");
13 #endif
```
(c)

Figure 7: Three categories of seeds: (a) external, (b) internal and (c) cross.

are prime targets for aspect refactoring, as they either disappear in the resulting application or are compiled in. In the advent of nested conditional compilation (C5), things get more complicated. The easiest technique would be to extract all the code found within the outermost conditional compilation guards into an advice body. However, the advice would then contain conditional compilation logic, and hence obfuscated program flow. Another approach would be to take into account (inclusion [8]) dependencies between conditional compilation flags and extract each possible path of preprocessor conditions into a single advice body. Each of these has an associated logic formula denoting the right combination of preprocessing conditions (Section 4.2). This approach may lead to code duplication if the outermost conditional compilation region contains unconditional code besides the inner conditional region, because every extracted advice body would contain a copy of this.

Finally, there is the hybrid category of "cross" seeds, i.e. seeds which are partially located in- and outside of conditional logic (Figure 7c). Naive refactoring into advice leads to advice bodies with conditional compilation within them. People should decide whether or not this is an improvement over the old situation. If not, the situation is similar to the one for "internal" seeds.

User interaction. At this point, the user should intervene. First, the miner should filter out false positives among the seeds, as all mining approaches have a precision of less than 100% [11]. One should also decide whether "internal" and "cross" seeds are desirable as is (with internal conditional compilation logic) or whether an extra finer-grained mining iteration should be performed to identify all possible conditional paths instead. Finally, humans should determine the specific (semantic) concern of the retained seeds (if any) and probably expand the candidate seeds into self-contained concern seeds [6]. Note that the name of preprocessor flags guarding conditional compilation regions does not necessarily correspond to the actual concern implemented by the conditional code. Ernst et al. [8] did assume this, but we have found some cases in the HotSpot VM where this was not the case.

Clearly, IDE support is indispensable in this step. More sophisticated support is required than traditional aspect mining techniques provide. Code with conditional compilation is hard to understand and so are "internal" and "cross" seeds. Even worse, some of the seeds could correspond to dead code if the particular configuration never occurs. It suffices to know the preconditions under which a seed normally is included in the build [13], i.e. the combination of (un)defined preprocessor flags to activate the seed. This knowledge enables the IDE to partially preprocess the source code such that only unknown flags are retained. Unselected code can be grayed out to enable users to better assess the seed and its associated program context [12, 17]. Of course, multiple sequences of preprocessor flag values can satisfy the precondition, so the human miner should be able to browse through all possible configurations or to at least understand the common configuration conditions.

This approach only works if one knows the exact preconditions under which conditionally guarded code is active. Latendresse [13] has proposed a fast ($O(SLOC)$) algorithm for this, which determines for every source code line the active conditions in terms of externally defined preprocessor flags. In addition, knowledge of actual builds could filter out infeasible configurations. Calculating this information via symbolic evaluation seems too expensive [13].

```
1 JNIEXPORT jlong JNICALL nogui() around Jp:
2  execution(Jp,``Java_sun_awt_..._getDisplay'')
3  && headless(_){
4   return NULL;
5 }
```

Figure 8: Advice extracted from Figure 3 (C2).

4.2 Aspect refactoring

Once suitable seeds have been identified and expanded, it is time to extract them into fully equipped advice [11] and to remove their traces in the base code. We discuss these issues in the context of preprocessor-driven source code.

Advice body extraction. In the case of behavioural crosscutting and with our assumptions in mind, body extraction boils down to the usual extraction process in preprocessor-unaware approaches. In practice, one needs a preprocessor-aware refactoring environment to perform the extraction with sufficient confidence [22, 9, 15]. It is known that subtle variations in the implementation of concerns make this step tedious, and will likely need refactoring of the code prior to extraction [4]. However, contrary to "external" seeds, "internal" and "cross" seeds usually are manifestations of tangling instead of scattering, because the preprocessor's fine-grainedness invites one-off adaptations or specific changes tailored for a particular configuration, similar to Colyer et al.'s notion of "heterogeneous concerns" [7]. The fine-grained usage patterns require new, lower-level AOP constructs to successfully apply aspects.

One frequent case is that various conditionally compiled code segments within the same function depend on common conditions, as if conditional logic is interleaved with the base code. A naive extraction would result in as many advice bodies as there are conditional segments. Temporal pointcut languages which allow to advise each step of a sequence would be better. Another possibility would be to coalesce all seeds from within the same function together in one advice and introduce a kind of labeled `proceed`-statement. Viability of such a multi-exit advice body is unknown.

Conditional branches in `switch`-statements are another interesting phenomenon (C3 on Figure 4). Converting these to advice is not straightforward. A dedicated `switch`-pointcut is very implementation-dependent, and the corresponding join points would be hard to identify. Advising the enclosing execution join point if its return value has a certain value, is not possible in general either. There does not seem to be a straightforward way to model this.

Even in the case of more "regular" crosscutting, extraction of advice is not trivial. In the example of Figure 3, the aspect miner has to decide which conditional branch represents the main concern. In the extracted advice[4] of Figure 8, we have opted for modularis-

[4] This advice is expressed in terms of the Aspicere aspect language for C [23].

ing the "headless" concern as an aspect. Hence, the around-advice (line 1) overrides the behaviour of the `execution` (line 2) of the procedure of Figure 3 by returning `NULL` (line 4). Optimisation of the woven code eliminates any run-time overhead in this case, such that the resulting code is equivalent to the preprocessed code.

For static crosscutting, quantified inter-type declaration is required, which exploits build-time configuration. Universal introduction of data fields (like in AspectJ) does not suffice.

Context extraction. Extraction of advice or data structure bodies is not enough, as the seeds' inputs and outputs have to be determined as well [2]. Preprocessor-driven source code introduces new challenges for this. Macro definitions can refer to any variable they want, as the exact references depend on the context in which the macro is expanded. This is actually "accidental" variable capturing, or some kind of dynamic scoping. When calculating the right context, the extraction algorithm should take macro definitions into account, possibly by implicitly expanding the macros in the seeds. A similar approach is needed when processing a seed containing file inclusion, and for conditional compilation. In the advice of Figure 8, no such context is needed. If the "non-headless" case had been extracted into the advice, the global `awt_display` should have been captured as context for the advice.

Pointcut deduction. Pointcuts of "external" and "cross" seeds can be derived via enumeration, logic induction, etc. For "internal" seeds, a very fine-grained, configuration-aware pointcut model is required. The first requirement follows from the theoretically unrestricted scope of preprocessor constructs. Any (part of a) statement can be conditionally compiled or used for a macro expansion. This even extends to type definitions. Being able to identify join points on such a small scale is crucial to succeed.

The requirement for configuration-aware pointcuts follows from the observation that an "internal" seed is identified by a physical location (actually a run-time event) and a logic formula which describes the specific configuration under which the seed would normally be compiled in. The former corresponds to the traditional notion of pointcut, but the latter is an extra refinement of the pointcut, determined exclusively by build-time configuration information. As such, pointcut languages should be able to take this information into account. Aspicere [23] e.g. is able to incorporate build knowledge as logic facts available to the pointcut and advice. As an example, the pointcut of Figure 8 (line 3) requires the `HEADLESS` compiler constant to be defined.

Removal of tangled code. The final step is to remove the original tangled code from the base code. In the example of Figure 3, only line 7 should be retained in the procedure body. Usually, this extraction step is performed manually, as automatic treatment is only possible if the seeds are uniform enough [4]. Only powerful refactoring support may be able to help.

There may be an easier way, however, in the form of partial evaluation of conditionally compiled code [3]. One could specialise the code in the areas where seeds have been extracted by (un)defining the relevant flags and leaving the others as is. Unneeded code would automatically disappear. This is not entirely bullet-proof, as unrelated code could be damaged if conditional flags would be preprocessed there by accident.

Just as in preprocessor-unaware systems, removal of tangled code and integration of the new aspects should be performed gradually in the presence of integration tests. Otherwise, new bugs could be attributed to either of both actions. Unfortunately, the configuration-dependent parts of the generated pointcuts in theory require thorough testing of all possible configurations. This is impossible to achieve in general, hence some clever heuristics are needed.

5. DISCUSSION

We can now comment on the fundamental question whether or not aspects are capable of replacing preprocessor code, and if they are, whether it is worth the effort. There are five important trade-offs one needs to consider before answering this question.

One of the most important factors is the code base itself, or rather the way in which the preprocessor is used. If disciplined, function-like macro expansion is used, or "inline seeds" cleanly encapsulate unconditional code, tools have a much better time because of the high degree of structure in the code. In the opposite cases, only manual approaches are viable.

Another trade-off regards AOP's promise of modularity. Advice is capable of modularising disciplined crosscutting conditional logic, while making the configuration conditions explicit. Given that conditional compilation frequently gives rise to tangling rather than scattering, this means that there will be many small one-to-one advices. Reuse of advice across multiple join points is less likely. This clearly defies the new modularity, as aspects become much too long and hence harder to understand.

Another factor compromising reuse are the finer-grained aspect constructs mentioned on many occasions in this paper. They indeed would make modeling preprocessor constructs easier, but at the same time they would introduce extra dependencies on implementation details in the base code. This leads to fragile pointcuts, but also reduces reuse opportunities. The only way to avoid this, is to program aspect-aware (non-oblivious) in the sense of keeping the code base as structured as possible without abusing the preprocessor. Of course, this has no effect in legacy systems, which de facto consists of heaps of existing code.

Preprocessor directives have some desirable properties which aspects do not share. Preprocessor code does not introduce any run-time penalty, nor does it inflate the resulting binary. The use of aspects on the other hand introduces a certain performance penalty in the build process [23] which can have a much larger impact than the fast string processing it replaces. As such, the increase in modularisation results in a sacrifice of build performance. Also, using current aspect technology, the build result (i.e. the woven application) will more than likely result in a bigger executable and slower run-time performance, although static optimisation techniques can eliminate many redundant instructions.

Finally, system developers are typically preprocessor experts, yet not familiar with the principles of AOP. One can expect a steep learning curve to adopt aspects and associated tools. Note that most macro refactoring tools are not fully automatic either, and also require user assistance [15]. Related to this trade-off is the question whether or not normal developers should be able to write aspects, or that they instead should follow special coding guidelines to trigger aspects which have been imposed by a select group of aspects experts. There is no clear answer yet.

To summarise, there are still many subproblems to be solved before people who are willing to give up the preprocessor for aspects will be able to do so.

6. CONCLUSION

We have explored the process of mining and extracting aspects in the context of preprocessor-driven systems, with examples from a large, real-world system. Conditional compilation in particular is amenable to aspect extraction. Unfortunately, the interaction, nesting and fine-grainedness of preprocessor directives complicate the identification of separable concerns and the subsequent extraction, and we have also observed the need for new AOP constructs. We have identified how duplication detection as well as preprocessor-

aware parsing are invaluable during aspect mining, and can rely on the many tools developed during research in preprocessing analysis and refactoring. Based on our exploration, we have distilled five trade-offs which give a tentative first idea about viability of replacing the preprocessor by aspects in a particular code base.

7. QUESTIONS

We do not propose one concrete technique, but consider aspect exploration, extraction and evolution in preprocessor-driven code in general. We need to mine unpreprocessed code to identify all seeds related to conditionally compiled concerns and macro expansions. Extraction of pointcuts could be done by calculating the conditions under which conditionally compiled code is (in)active. Advice extraction needs to take into account free variables, macro definitions, etc. Guaranteeing behaviour preservation is difficult, because all feasible build configurations should be tested. The fragile pointcut problem is a concrete risk because of the fine-grainedness of preprocessor directives. However, we believe that our technique leads to better evolvable base code because it becomes decoupled from the complex conditional compilation flow.

Acknowledgements

Bram Adams is supported by a BOF grant from Ghent University.

8. REFERENCES

[1] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter. Design recovery and maintenance of build systems. In *Proc. of the 23rd International Conference on Software Maintenance (ICSM)*, Paris, France, October 2007.

[2] E. L. A. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proc. of the 20th International Conference on Software Engineering (ICSE)*, pages 64–73, Kyoto, Japan, April 1998.

[3] M. Baxter, I.D.; Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE)*, pages 281–290, Stuttgart, Germany, 2001.

[4] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *Proc. of the 6th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 199–211, Vancouver, BC, Canada, March 2007.

[5] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelen. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005.

[6] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Control*, 14(3):209–231, 2006.

[7] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proc. of the 3rd international conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65, Lancaster, UK, 2004. ACM.

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

[9] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Aug. 2005.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[11] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, IV(LNCS 4640):143–162, 2007.

[12] B. Kullbach and V. Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *Proc. of the Eighth Working Conference on Reverse Engineering (WCRE)*, page 3, Stuttgart, Germany, 2001. IEEE Computer Society.

[13] M. Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proc. of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, page 170, Benevento, Italy, 2003. IEEE Computer Society.

[14] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. of the 11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141, Delft, The Netherlands, November 2004.

[15] B. McCloskey and E. Brewer. ASTEC: a new approach to refactoring C. *SIGSOFT Softw. Eng. Notes*, 30(5):21–30, 2005.

[16] C. A. Mennie and C. L. Clarke. Giving meaning to macros. In *Proc. of the 12th IEEE International Workshop on Program Comprehension (IWPC)*, pages 79–85, Bari, Italy, 2004. IEEE Computer Society.

[17] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, Vancouver, BC, Canada, 2007.

[18] S. S. Somé and T. C. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Proc. of the 6th International Workshop on Program Comprehension (IWPC)*, page 118, Ischia, Italy, 1998. IEEE Computer Society.

[19] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C News. In R. Adams, editor, *Proc. of the USENIX Conference*, pages 185–198, Seattle, WA, US, June 1992. USENIX Association.

[20] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng.*, 29(11):1019–1030, 2003.

[21] L. Vidács, A. Beszédes, and R. Ference. Columbus schema for C/C++ preprocessing. In *Proc. of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR)*, page 75, Tampere, Finland, 2004. IEEE Computer Society.

[22] M. Vittek. Refactoring browser with preprocessor. In *Proc. of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, page 101, Benevento, Italy, 2003. IEEE Computer Society.

[23] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *Proc. of the 10th European Conference on Software Maintenance and Reengineering (CSMR)*, Bari, Italy, 2006.