Can we Refactor Conditional Compilation into Aspects?

Bram Adams[1] Wolfgang De Meuter[2] Herman Tromp[1] Ahmed E. Hassan[3]
bram.adams@ieee.org wdmeuter@vub.ac.be herman.tromp@ugent.be ahmed@cs.queensu.ca

[1]GH-SEL, INTEC, Ghent University (Ghent, Belgium)
[2]PROG, DINF, Vrije Universiteit Brussel (Brussels, Belgium)
[3]SAIL, School of Computing, Queen's University (Kingston, ON, Canada)

ABSTRACT

Systems software uses conditional compilation to manage crosscutting concerns in a very fine-grained and efficient way, but at the expense of tangled and scattered conditional code. Refactoring of conditional compilation into aspects gets rid of these issues, but it is not clear yet for which patterns of conditional compilation aspects make sense and whether or not current aspect technology is able to express these patterns. To investigate these two problems, this paper presents a graphical "preprocessor blueprint" model which offers a queryable representation of the syntactical interaction of conditional compilation and the source code. A case study on the Parrot VM shows that preprocessor blueprints are able to express and query for the four commonly known patterns of conditional compilation usage, and that they allow to discover seven additional important patterns. By correlating each pattern's potential for refactoring into advice and each pattern's evolution of the number of occurrences, we show that refactoring into advice in the Parrot VM is a good alternative for three of the eleven patterns, whereas for the other patterns trade-offs have to be considered between robustness and fine-grainedness of the advice.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Aspect-oriented programming; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.3.4 [**Processors**]: Preprocessors

General Terms

Documentation, Experimentation, Measurement

1. INTRODUCTION

Systems software like operating systems, compilers, virtual machines, etc. is typically developed in C or C++, and combines heavy C preprocessor usage with complex build system trickery to handle configuration [34]. Conditional compilation and preprocessor flag primitives are mixed into the source code to tailor the software to different platforms and to enable sophisticated selection of features. This makes these systems hard to understand and maintain [14, 32].

The C preprocessor just manipulates text, and ignores C's syntax rules. Ernst et al. [12] have measured that on average 4% of all lines of source code contains conditional compilation directives which together control 37% of the source code, especially for dealing with program portability. A considerable part of the source code is coupled directly with the build system for the purpose of configuration, and, in addition, the preprocessed program can be one of a multitude of C programs. Hence, the C preprocessor is an important source of errors and of confusion [14, 32]. Nevertheless, every C and even C++ [27] programmer has to understand how the C preprocessor interacts with a system [14].

Many researchers [2, 26, 30, 31] propose to replace conditional compilation with aspects (AOP) [20], as a conditional region typically corresponds to a scattered and tangled implementation of a crosscutting concern like debugging, tracing, platform-dependent logic, etc. Refactoring such a region into explicit advice with the region's condition as the pointcut promises a more semantic and better maintainable description of the intent of the conditional regions [16], as it makes the base code independent from the preprocessor and the build system. A conditional region at the start of a procedure body could e.g. be refactored into `before`-advice on the procedure's execution join point. As the build configuration is statically known, the aspect weaver can make the woven code as efficient as the preprocessed code. However, it is not clear yet whether AOP's promise of improvement holds for all patterns of conditional compilation usage, and if so, whether aspect languages can express these patterns.

There are four patterns of conditional compilation commonly found in literature [2, 12, 27, 30], but these especially cover straightforward use cases. To be able to express other important patterns which could benefit from refactoring into advice and to check for their relevance by finding all their occurrences, this paper derives three requirements for a queryable model of the interaction of source code and conditional compilation, and proposes a realisation of this model, named "preprocessor blueprint". This models how conditional compilation and base constructs are nested and ordered relative to each other within a given file. In addition, it provides a declarative means to specify blueprint patterns of conditional compilation usage and to find all occurrences of these patterns.

This paper applies preprocessor blueprints to find out whether or not the refactoring of conditional compilation patterns into advice is technically feasible, independent from the semantics [12, 22] or purpose [16] of the conditional code. Blueprint patterns are used to express the four commonly found patterns of conditional compilation [2, 12, 27, 30], and to analyse the blueprint model of a representative piece of systems software, i.e. the Parrot VM (vir-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-442-3/09/03 ...$5.00.

tual machine) [28], for seven additional patterns. For each pattern, the possible strategies for refactoring into advice are discussed and the number of its occurrences in each Parrot VM release is calculated. This information determines which patterns in the Parrot VM are the most important and which ones are only temporarily used. We have found that three of the eleven identified blueprint patterns directly benefit from refactoring into advice. For the other patterns, there are important trade-offs to consider. Conditional compilation often is the best implementation strategy.

This paper makes the following contributions:

- Three requirements for a model of the syntactical interaction of conditional compilation and the source code are derived from the four known patterns of conditional compilation, and from the join point models of aspect languages for C.
- The preprocessor blueprint model realises these three requirements, and we show how declarative patterns of conditional compilation usage can be expressed and queried with it.
- To validate the model's ability to determine the potential of refactoring conditional compilation into advice, all releases of the Parrot VM are queried for the four known patterns, and seven additional patterns are identified. For each pattern, the potential for refactoring into advice is analysed.
- The evolution of the distribution of the number of occurrences of the eleven patterns is investigated throughout the Parrot VM release history. This provides insight into the importance of each pattern and their stability over time. With this information, the potential for refactoring conditional compilation into advice is determined for the Parrot VM.

Section 2 first gives the necessary background on the C preprocessor. Then, Section 3 presents the four commonly known patterns of conditional compilation, and analyses how they can be refactored into advice. From this, three requirements for a model of conditional compilation usage are derived (Section 4.1). Preprocessor blueprints realise these three requirements and can be queried declaratively (Section 4.2). In Section 5, blueprints are applied to the Parrot VM to validate their ability to identify additional patterns of conditional compilation usage. For each such new pattern, we analyse whether or not it can be refactored into advice. The evolution of the number of occurrences of the eleven discussed patterns is investigated in Section 6 to determine the impact and nature of each pattern in the Parrot VM. Finally, our results are compared to related work (Section 7), future work is discussed (Section 8) and the findings of this paper are summarised (Section 9).

2. THE C PREPROCESSOR

The C preprocessor is a text processing tool that complements the C compiler to resolve many shortcomings of C [19] and even of C++ [27]. The three major constructs it provides, are [12]:

`#include` textual file inclusion
`#define/#undef` macro or constant (un)definition
`#ifdef` conditional compilation

Developers typically use file inclusion to separate declarations from the implementation modules. A C macro is a syntactic substitution mechanism for defining syntactic sugar, inline functions, constants, etc. Each macro occurrence is expanded into its value by the C preprocessor. A degenerate macro which corresponds to a constant or is only known to be (un)defined is called a "preprocessor flag" or "constant", and is used in the logical condition of a conditional compilation region. Such a region encloses source code lines which are only compiled if the region's condition is satisfied. Figure 3 shows a C code example with a conditional region on lines 2–4. If the `THREAD_DEBUG` flag is defined, the preprocessor retains the code on line 3 and removes lines 2 and 4. Otherwise, lines 2–4 all disappear. The three major preprocessor constructs are the primary mechanism for a build system to control the variability points in C/C++ source code [1, 34].

There exists a clear link between conditional compilation and aspects. The `THREAD_DEBUG` condition on line 2 of Figure 3 e.g. can be interpreted as a pointcut, the conditional code on line 3 as advice, and the physical position of the conditional region (i.e. the start of the `pt_transfer_sub` procedure body) as the join point shadow[1] [18] at which the advice should match. The major difference with advice is that a conditional region's condition is statically evaluated by the C preprocessor, whereas aspects conceptually are woven at run-time. However, if aspect weavers statically exploit the known preprocessor flags, aspects are as efficient as conditional compilation [1]. Next, we analyse how four commonly found patterns of conditional compilation can be refactored into advice.

3. WELL-KNOWN PATTERNS OF CONDITIONAL COMPILATION

This section presents the four patterns of conditional compilation which are commonly known [2, 26, 30, 31], with two goals in mind. First, we want to analyse whether or not they can be refactored into advice. Second, Section 4.1 derives requirements from them for a model of syntactical interaction between conditional compilation and the base code, which can be queried to detect interesting interactions from the perspective of refactoring into aspects.

The four known patterns of conditional compilation usage can be categorised as "Coarse-grained Conditional Compilation" (Section 3.1) or as "Fine-grained Conditional Compilation" (Section 3.2). For most of them, an example code snippet is given and the potential for refactoring into advice is discussed. The bold entries in Table 1 summarise these patterns' alternative AOP implementations.

3.1 Coarse-grained Conditional Compilation

Coarse-grained Conditional Compilation patterns make a whole definition conditional, similar to structural crosscutting concerns.

3.1.1 Multiple Inclusion Protection

Multiple Inclusion Protection is a well-known idiom [12, 27] in C/C++ header files to ensure that the preprocessor textually includes a header file at most once in order to avoid duplicate declarations and definitions. The first time the header file is included, a particular preprocessor flag is defined. From the second time on, the region's condition fails because of this defined preprocessor flag, and hence the region is not included anymore.

This pattern represents a crosscutting meta-concern, i.e. a support idiom for the preprocessor [19]. It cannot be implemented conveniently with source code-level aspects. Instead, the build system could automatically transform each header file before compilation. However, this means that the header files are only portable the build system is ported as well. Trusting on the intelligence of new compilers causes similar problems. Hence, conditional compilation is the only portable implementation of Multiple Inclusion Protection.

3.1.2 Conditional Definition

The pattern in Figure 1 captures the conditional definition of type, procedure and macro definitions [12, 30]. It is often used to decide between alternative versions of a type, procedure or macro. If a developer does not want debugging output, the trace macro ex-

[1]For each run-time join point, a corresponding construct in the source code exists which is executed by the join point at run-time. E.g., an actual procedure call forms the shadow of a `call` join point. Shadows are used by compile-time weavers to statically weave aspects.

```
1 #ifdef NDEBUG
2 #  define TRACE_FM(i, c, m, sub)
3 #else
4 static void
5 debug_trace_find_meth(. . .){
6   . . .
7 }
8 #  define TRACE_FM(i, c, m, sub) \
9    debug_trace_find_meth(i, c, m, sub)
10 #endif
```

Figure 1: Conditional Definition in src/objects.c.

pands to nothing (line 2 of Figure 1). Otherwise (lines 8–9), the macro calls the procedure on lines 4–7.

The easiest way to decouple the base code from this pattern is to refactor the conditionally defined code into a separate base code module [32], and to translate the region's condition into build system logic. However, developers need to have a good understanding of how the build system works and how it interacts with the source code. Worse, with increasing nesting of conditional regions, the number of refactored modules increases combinatorially and the correlation between conditional branches becomes blurred [24, 31]. Hence, refactoring into modules is not always viable.

A second technique uses pointcuts to select whether or not the weaver should include a type or procedure definition, instead of making this the responsibility of the build system. Although such quantified inter-type declaration[2] (ITD) exists [21, 30], it is not a common feature. In addition, there should be pointcut primitives to access the build configuration [1, 30], i.e. the defined preprocessor constants and selected source code modules.

A third technique, which does not work for data types, is to extract the body of a procedure into advice such that `around`-advice can redefine the empty procedure based on the build configuration. However, using a behavioural crosscutting mechanism for a structural crosscutting concern requires extra boilerplating, and might introduce run-time and binary size overhead. Fortunately, the conditions in the advice's pointcut only depend on the static build configuration. To summarise, quantified ITD is a suitable AOP implementation strategy for Conditional Definition, but the others are either not feasible, or only feasible in certain situations.

3.2 Fine-grained Conditional Compilation

The second class of conditional compilation considers intra-procedure and intra-type patterns.

3.2.1 Conditional Signature

```
1 #if (defined __STDC__ || . . .)
2 static int YYID(int i)
3 #else
4 static int YYID(i)
5 int i;
6 #endif
7 {
8         return i;
9 }
```

Figure 2: Conditional Signature in compilers/.../pirparser.c.

The pattern in Figure 2 selects between an ANSI and K&R [19] procedure signature [12]. Just like Multiple Inclusion Protection (Section 3.1.1), Conditional Signature is a crosscutting meta-concern. It tackles variability across compilers.

[2] Inter-type declaration adds fields and methods to data structures or classes.

A possible implementation technique is to make the build system transform each procedure signature before compilation, but this again causes portability problems. Contrary to Multiple Inclusion Protection, the variants of a procedure definition can be refactored into separate modules or introduced by quantified ITD, but these approaches duplicate source code. Conditional compilation is the best implementation strategy for Conditional Signature.

3.2.2 Simple Conditional Compilation

```
1 PMC* pt_transfer_sub(. . .){
2 #if THREAD_DEBUG
3   PIO_eprintf(s,. . .);
4 #endif
5   return make_local_copy(d, s, sub);
6 }
```

Figure 3: Simple Conditional Compilation in src/thread.c.

```
1 PMC* pt_transfer_sub(Parrot_Interp d,
2                      Parrot_Interp s,PMC *sub){
3   return make_local_copy(d, s, sub);
4 }
5
6 void debug_transfer(Parrot_Interp S, PMC* Sub)
7      before Jp:
8  execution(Jp,``pt_transfer_sub'')
9  && args(Jp,[_,S,Sub])
10 && thread_debug(_){
11   PIO_eprintf(S, "copying over subroutine [%Ss]\n",
12               Parrot_full_sub_name(S, Sub));
13 }
```

Figure 4: Aspect implementation of Figure 3.

Simple Conditional Compilation contains one conditional block at the start and/or end of a procedure or data type. Reynolds et al. [30] found that 45% of configuration-dependent code in procedure bodies (24% of all conditional code) and 52% of `struct` fields definitions adhere to this pattern. Figure 3 gives an example of a conditional region at the beginning of a procedure body.

Intuitively, Simple Conditional Compilation can be replaced by `before`-, `after`- or `around`-advice on the `execution` of the procedure, or quantified ITD [30]. Figure 4 shows a `before`-advice for Figure 3 in the Aspicere AOP language for C [1]. The conditional code has moved from the base code (lines 1–4) to the advice body on lines 11–12. The advice's pointcut (lines 8–10) is expressed in terms of the build configuration (`thread_debug` on line 10) of Figure 3 (`THREAD_DEBUG` on line 2), which is communicated to Aspicere's weaver right before weaving starts. Hence, the build system is now coupled to the advice instead of to the base code. As the build configuration is completely known statically, the weaver can eliminate run-time checks in the woven code.

An important issue with the refactoring into advice is that the state used by the non-conditional part of a procedure body (`d`, `s` and `sub` on line 5 of Figure 3) should be accessible to the advice as join point context. This is not always easy, as it might comprise local variables. A second important issue is that the precedence of the refactored advices or the introduction of `struct` fields is hard to get right in case of multiple conditional regions or nesting. The original preprocessor code can deal better with this. Hence, there is no clear-cut AOP implementation technique for Simple Conditional Compilation. This wraps up the analysis of the four commonly known patterns of conditional compilation.

4. PREPROCESSOR BLUEPRINTS

This section derives three requirements for a model of syntactical patterns of conditional compilation usage, and presents the design (Section 4.2) and implementation (Section 4.3) of preprocessor blueprints, which realise these requirements. Preprocessor blueprints provide a declarative, graphical way to document and query conditional compilation usage. As such, this model supports the analysis of refactoring conditional compilation into aspects.

4.1 Requirements

To analyse the syntactical interaction of conditional compilation and normal source code, a relatively simple model of conditional compilation suffices. Existing models of preprocessor usage are too complex because they try to encompass all preprocessor constructs in whichever way they interact with a program. This is necessary for refactoring environments [5, 13, 17, 27, 37] or compilers [36], but this paper only focuses on conditional compilation. Moreover, some models treat the normal C code as text [13, 36]. This is undesirable as well, because we are interested in the specific interaction of conditional compilation with procedure and type definitions, procedure calls, global variable access, etc., as the patterns in Section 3 have shown. These observations suggest that a simple model of conditional compilation and source code suffices.

Furthermore, it does not make sense to spend excessive attention to "undisciplined" preprocessor usage which breaks C's syntax rules, as they intuitively are less likely to be refactorable into advice, and they do not occur that often in practice either. Ernst et al. [12] e.g. point out that roughly two thirds of preprocessor usage corresponds to simple, disciplined patterns, Vittek [37] notes that "usually #if directives do not break the structure of source code" and Baxter et al. [5] 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". Hence, this suggests that too fine-grained interaction patterns can be disregarded by our model.

Based on these observations and the patterns of Section 3, three requirements for a model of the syntactical patterns of conditional compilation usage were derived and are discussed below:

1. The model should focus on conditional regions, procedure and data definitions, procedure calls and global variable access. The uninteresting ("opaque") statements should not be interpretable individually, but instead coalesced into an "opaque" sequence of program statements.
2. The model should explicitly depict the nesting of model elements in conditional regions or procedure/data definitions, and for each nesting relation it should show how the nested model elements are ordered relative to each other.
3. The model should be complemented by a declarative pattern matching facility to find all occurrences of a particular conditional compilation usage pattern.

4.1.1 Focus on Interesting Statements

Depending on the join point model of an aspect language, some program statements are more interesting than others. Aspect languages [1] for C/C++ fancy procedure call and execution join points, global variable access join points, ITD of data fields and definitions, and ITD of procedure definitions. The shadows of these join points correspond to procedure and type definitions, files, procedure calls and global variable references. A language with statement-level join points [11] would also be interested in assignments, pointer dereferences or even macro expansions. However, no aspect language for C/C++ supports this yet, the patterns of Section 3 do not need this, and robust pointcuts are hard to define. Hence, our model should only focus on the few interesting statements.

Still, the uninteresting program statements must not be dismissed. As these do not have a corresponding join point, they make it hard to write a robust pointcut for conditional regions. Hence, the model should record the presence of these seemingly uninteresting ("opaque") sequences of statements and their position relative to the non-opaque statements. Any uninteresting AST node should be replaced by an opaque statement, and subsequent opaque statements should be coalesced. This is similar to Krone et al. [22], who abstract source code statements into blocks of consecutive lines of code.

To summarise, a model of the syntactical interaction of conditional compilation and the source code should focus on the program statements which correspond to join points in the used aspect language, and keep track of the presence of uninteresting statements.

4.1.2 Nesting and Ordering of Model Elements

The second requirement emphasises nesting and ordering as the primary relations between the model's elements. This follows from the concept of conditional compilation as well as from the patterns in Section 3. First, a conditional region is either enclosed completely by another conditional region, or not at all ("inclusion dependence" [12]). Second, the main difference between the two coarse-grained (Section 3.1) and fine-grained patterns (Section 5.3.5) is that the former enclose high-level program statements, whereas the latter are nested within a procedure body. Hence, nesting is a defining characteristic of conditional compilation.

To reason about patterns of conditional compilation usage, the ordering of the model elements relative to each other is crucial. For Simple Conditional Compilation (Section 3.2.2) e.g., it does matter whether a conditional region appears at the start or end of a definition, or is preceded by an opaque piece of code. Hence, nesting and ordering should both be represented.

4.1.3 Declarative Pattern Matching Facility

To determine the impact of a pattern of conditional compilation, a declarative pattern matching facility is needed. Most tools hardcode a set of patterns [30], or only provide an imperative way to specify and detect occurrences of patterns. PCp^3 [4, 12] e.g. is an extensible C preprocessor based on callbacks. Occurrences of individual preprocessor constructs are easy to handle, but detection of sequences (second requirement) requires the developer to manually implement a state machine. In addition, knowledge about Pcp^3's internals is needed to write the callbacks. A declarative approach is able to circumvent these limitations.

Mennie et al. [27] do have a fact base representation of the source code, but do not use it to find the occurrences of patterns of conditional compilation usage. As such, they are limited to simple patterns of conditional compilation usage (e.g. regions with "1" or "0" as condition). A declarative querying approach enables to express and query for more complex interactions.

4.2 Preprocessor Blueprint Model

This section presents the preprocessor blueprint model, which realises the three requirements of Section 4.1. To illustrate the expressiveness of the model, we use it to express the four patterns discussed in Section 3. Afterwards, the next section shows how the preprocessor blueprints enable to express and query for seven additional, more complex patterns of conditional compilation.

4.2.1 Focus on Interesting statements

Figure 5 shows example code adapted from the Parrot VM and the corresponding blueprint model. This is a tree-like graph with six different kinds of nodes and two kinds of edges.

As dictated by the first requirement, preprocessor blueprints have

```
1  void Parrot_setenv(. . . name,. . . value){
2  #ifdef SETENV
3    my_setenv(name, value, 1);
4  #else
5    int name_len=strlen(name);
6    int val_len=strlen(value);
7    char* envs=glob_env;
8    if(envs==NULL){
9      return;
10   }
11   strcpy(envs,name);
12   strcpy(envs+name_len,"=");
13   strcpy(envs+name_len + 1,value);
14   putenv(envs);
15 #endif
16 }
17
18 #ifdef LINUX
19 extern int Parrot_signbit(double x){
20     union{
21         double d;
22         int i[2];
23     } u;
24     u.d = x;
25 #  ifdef BIG
26     return u.i[0] < 0;
27 #  else
28     return u.i[1] < 0;
29 #  endif
30 }
31 # endif
```

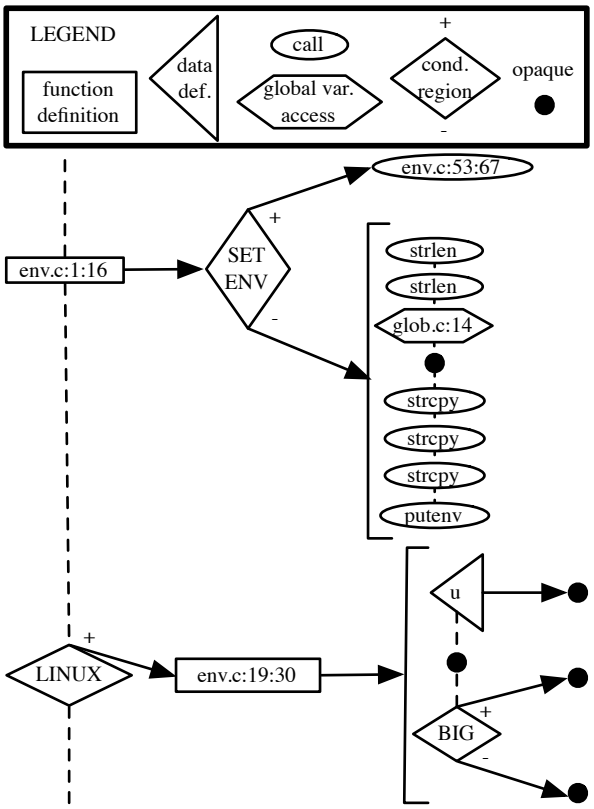

Figure 5: Beginning of an example source file ("env.c") adapted from Parrot VM 0.6.2 (a), and its corresponding blueprint (b).

nodes for procedure definitions (rectangle), data definitions (triangle), procedure calls (ellipse), global variable access (hexagon) and conditional regions (diamond). Opaque code fragments are abstracted into a black dot. Procedure and data definitions are labeled by the file they are part of and the start and end line numbers. Procedure calls and global variable accesses are labeled by the procedure definition or global variable they are referring to, or just by the name of a system procedure or of a function pointer variable. Conditional nodes are labeled with their condition.

4.2.2 Nesting and Ordering of Model Elements

The two kinds of edges implement nesting and ordering between a blueprint's nodes. Procedure and data definitions, and conditional regions can be the source node of a (plain) nesting edge, but never a leaf node. Procedure calls, global variable access and opaque nodes can only be leaf nodes. A conditional node has at most two outgoing nesting edges. An edge starting from the top of such a node represents a positive conditional check, i.e. `#ifdef` or the `#else`-clause of an `#ifndef` condition, whereas an edge from the bottom is just the opposite. Nested regions and regions with multiple branches are modeled by nesting diamond nodes. Nesting edges provide a crisp overview of nesting in a blueprint.

The dashed lines model the ordering between sibling model elements. The upper elements lexically appear first to make the ordering of elements relative to each other intuitively clear.

4.2.3 Declarative Pattern Matching Facility

The preprocessor blueprint pattern matching facility is best described by means of examples. Figure 6a, Figure 6b, Figure 6e and Figure 6f contain declarative patterns that correspond to the four patterns of conditional compilation usage of Section 3. These patterns contain the same kinds of nodes as the preprocessor blueprint model, enhanced by those shown in the legend of Figure 6.

The additional nodes are illustrated on Figure 6a and Figure 6b. The former matches a conditional region at the file-level which includes zero or more model elements (star node), whereas the latter picks out regions which enclose at least one procedure or data definition (trapeze node). The pattern of Figure 6a is said to be in "absolute mode", because it has to match exactly with the whole preprocessor blueprint (one source file), similar to the use of `^` and `$` in regular expressions. On the other hand, the pattern of Figure 6b can match anywhere inside a blueprint, i.e. it is in "relative mode". Graphically, the outer left node of a relative mode pattern like Figure 6b has a dashed edge through it, whereas the same node in an absolute mode pattern (just Figure 6a) does not.

A nesting edge which starts from the middle of a conditional node matches any conditional check, whether it is positive or negative. Because patterns frequently have to check for the occurrence of either a procedure or data definition, there is a dedicated trapeze node for this. Similarly, a rounded rectangle denotes either a procedure call or global variable access.

Figure 6e and Figure 6f show two more complex, fine-grained patterns. The former selects those patterns where code of a particular procedure initially is nested inside two branches of a conditional compilation, and later appears independently again. This corresponds to a Conditional Signature pattern (Section 3.2.1), i.e. only the procedure's signature depends on the build configuration. The procedure definition nodes are labeled identically (`p`) to en-

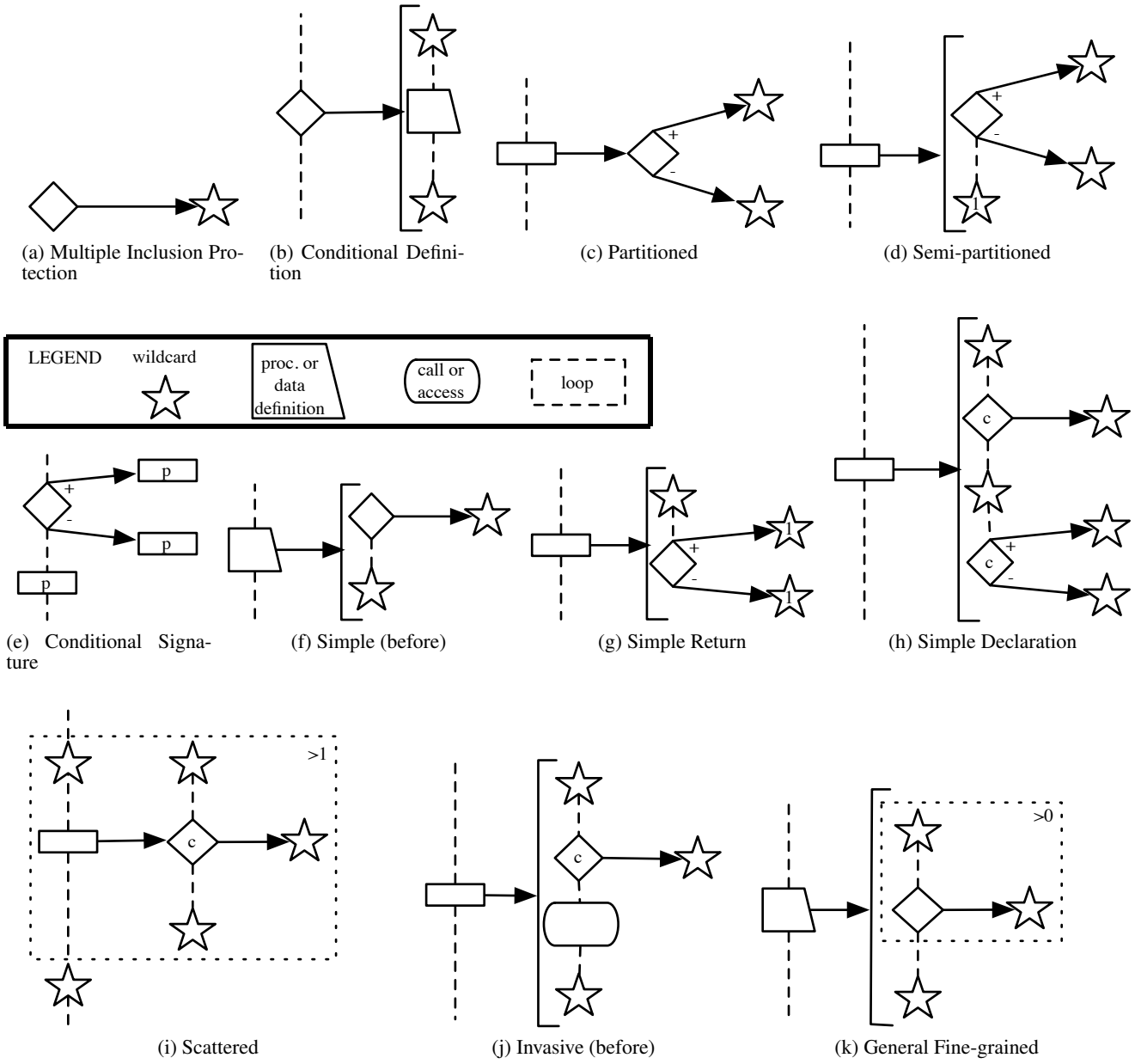

Figure 6: Declarative blueprint patterns for all discussed patterns of conditional compilation usage.

force that all of them correspond to parts of the same procedure. Figure 6f selects definitions with as first construct a conditional region. This is the `before` case of Simple Conditional Compilation.

As already mentioned, the scope of pattern matching in a blueprint model is determined by the pattern's mode (relative or absolute). A pattern in relative mode is not only matched on the top-level of a file, but also at all lower (nested) levels. This is necessary to detect combinations of patterns, e.g. a Conditional Definition in a file with Multiple Inclusion Protection, or Simple Conditional Compilation inside a Conditional Definition. The influence of nesting on refactoring into advice is that the advice's pointcut is conjuncted with an extra build configuration-dependent check.

To summarise, preprocessor blueprints can declaratively express Section 3's patterns of conditional compilation usage. After a discussion of a prototype implementation, Section 5 shows additional patterns of conditional compilation we discovered in a case study.

4.3 Implementation

We built a prototype implementation ("R3V3RS3"[3]) for the blueprint model and pattern matching facility, based on a robust C parser and regular expression matching. The source code first is pretty-printed using "uncrustify" [35] and stripped from comments and blank lines by "cloc" [8], in order to improve the results of later phases. The pretty-printed version is then parsed by Fetch [15]. This is a reverse-engineering tool chain based on robust C/C++ fact extraction [10] and a lightweight parser. Its output is a graph in the Rigi Standard Format (RSF) that describes the program AST and

[3] Pronounced as "reverse".

preprocessor usage. We use Prolog and Perl scripts to generate a textual representation of preprocessor blueprints from the RSF file.

The textual representation of the preprocessor blueprints can be queried for occurrences of specific patterns of conditional compilation via Perl regular expressions which are generated from the graphical blueprint patterns. R3V3RS3 finds all occurrences of a pattern in a preprocessor blueprint, and removes these patterns from the blueprint if desired. Patterns in relative mode are applied to all levels of nesting in the blueprint. R3V3RS3 gives as output the number of occurrences of each pattern as well as the filtered preprocessor blueprint. The next section uses R3V3RS3 to discover additional patterns of conditional compilation usage.

5. PARROT VM CASE STUDY

To validate the preprocessor blueprints' ability to express and query for all occurrences of a conditional compilation pattern, we have applied R3V3RS3 to the Parrot VM [28]. This is a relatively young, open source VM, with a well thought-out development process [29]. Its preprocessor usage is based on best practices, so our measurements correspond to a best-case scenario from the perspective of aspect refactoring, i.e. an upper bound on the potential for refactoring into advice. Experiments on systems with excessive preprocessor (ab)use is considered future work. The next section presents the case study approach, followed by a discussion of seven new patterns of conditional compilation usage discovered with the preprocessor blueprint model.

5.1 Approach

We have studied the Parrot VM in two steps. First, the preprocessor blueprints for the last known version of the Parrot VM were analysed to discover the number of occurrences of the four patterns of syntactical interaction discussed in Section 3. We then iteratively used R3V3RS3's ability to filter occurrences of matched patterns out of a blueprint in order to visually spot interesting patterns in the remaining blueprint, specify them as a blueprint pattern and re-apply R3V3RS3. For each pattern, possible implementation strategies were analysed. These are summarised in Table 1 and explained in the next sections. By carefully choosing the order of matching the patterns, we avoided that general patterns incidentally would match more specific patterns. The newly identified patterns are categorised similar to Section 3, whereas conditional compilation usage which is either hard or not useful to refactor into advice is grouped under "Ad Hoc Pattern"s (Section 5.4). In future pattern mining efforts, the set of collected conditional compilation patterns can be used to filter out known patterns in advance.

The second step in our approach analyses across all public releases of the Parrot VM[4] the evolution of the number of occurrences of the patterns detected in the first step for the last Parrot VM version. This gives insight into the importance of the identified patterns of conditional compilation in practice as well as into the stability of the patterns over time. If we combine the former with the results of Table 1, this tells us whether or not the patterns which are easy to refactor into advice are also the most important ones in practice. The stability of the patterns over time teaches us which patterns remain important over time and which ones are only used temporarily. Those patterns are not important to refactor into aspects. The next sections present the results of the first step in the case study. Section 6 discusses the second step.

[4]We dropped the five versions between half of May, 2007 and half of December, 2007 because of technical issues.

5.2 Coarse-grained Conditional Compilation

In addition to the two patterns of Section 3.1, two additional patterns of Coarse-grained Conditional Compilation were found.

5.2.1 Partitioned Conditional Compilation

```
1  . . . fetch_iv_le(INTVAL w){
2  #if !PARROT_BIGENDIAN
3    return w;
4  #else
5  #  if INTVAL_SIZE == 4
6    return (w << 24) | ((w & 0xff00) << 8) |
7           ((w & 0xff0000) >> 8) | (w >> 24);
8  #  else
9    INTVAL r;
10
11   r = w << 56;
12   . . .
13   return r;
14 #  endif
15 #endif
16 }
```

Figure 7: Partitioned Conditional Compilation in src/byte-order.c.

Partitioned Conditional Compilation encloses a procedure's body instead of its definition (Figure 7). Typically, there are two conditional branches, but each of them may contain further nested regions. A blueprint pattern for Partitioned Conditional Compilation with two branches (the most common case) is shown on Figure 6c.

A similar refactoring approach can be used as for Conditional Definition, with an important difference. Instead of deciding whether or not a procedure definition should be compiled, the definition is always included, but with a different body depending on the build configuration. `around`-advice on an empty definition is a perfect fit for this pattern. Because the build configuration is statically known, the aspect weaver can optimise away any run-time overhead. Alternatively, quantified ITD is possible, but this requires a separate ITD construct for every branch in the procedure body. Note that the Parrot VM contains a number of procedures with Partitioned Conditional Compilation that are almost identical, except for the fact that the logical conditions are inverted. This is a perfect case for advice which can be bound to multiple pointcuts.

5.2.2 Semi-partitioned Conditional Compilation

Semi-partitioned Conditional Compilation is similar to Partitioned Conditional Compilation, except that the procedure body's last line, typically a return statement, is not conditional. This is expressed by the pattern on Figure 6d. The lower star node is labeled with "1" to enforce that there is only one final statement in the procedure body. Although this statement could just as well be a nested conditional region, we have identified that in practice all occurrences of this pattern in the Parrot VM have only one statement at the end.

The unconditional final statement can be distributed over the conditional blocks to enable the AOP solutions for Partitioned Conditional Compilation, with the same benefits and problems. Alternatively, the conditional logic could be refactored into `before`-advice. However, this requires that the state which is referenced by the final statement can be accessed and manipulated by the `before`-advice. This could be a local variable declared by the conditional regions. It is not clear how the `before`-advices for each conditional branch should access this state in a clean way.

To summarise, except for Multiple Inclusion Protection, the four discussed patterns of Coarse-grained Conditional Compilation patterns in principle can be refactored into build system-controlled

Table 1: Summary of the possible AOP refactorings of the four well-known (in bold) and seven newly identified patterns of conditional compilation usage. A +/- means that the technique clearly works/fails, whereas other entries have trade-offs to be considered.

	AOP refactoring → ↓ pattern	modules	quantified ITD	around empty execution	around execution	before execution	after execution	call advice
coarse-grained	**multiple inclusion**	-	-	-	-	-	-	-
	conditional definition	• build knowledge needed • combinatorial module increase	+	• larger binary • not for data	-	-	-	-
	partitioned		duplication	+	-	-	-	-
	semi-partitioned			+	-	join point context	-	-
fine-grained	**cond. signature**	duplication		-	-	-	-	-
	simple	-	precedence	-	• fine-grained join point context passing needed • precedence of conditional regions at same join point			-
	simple return	-	-	-	-	-	• choose one region as base code • join point context	-
	simple declaration	-	-	-	join point context	-		-
	scattered	-	-	-	-	-	-	• heterogeneity • prepare base code
	invasive	-	-	-	-	-	-	join point context
	general fine-grained	-	-	-	-	-	-	• fragile advice • prepare base code

source modules, quantified ITD and/or `around`/`before`-advice. Each has its benefits and drawbacks, however, and conditional compilation sometimes is the best implementation choice.

5.3 Fine-grained Conditional Compilation

We have found five additional patterns of Fine-Grained Conditional Compilation on top of those of Section 3.2.

5.3.1 Simple Return

```
1  static opcode_t fetch_op_be_4(. . .){
2    . . .
3  #if PARROT_BIGENDIAN
4  #  if OPCODE_T_SIZE == 8
5    return u.o >> 32;
6  #  else
7    return u.o;
8  #  endif
9  #else
10 #  if OPCODE_T_SIZE == 8
11   return (opcode_t)(fetch_iv_le((INTVAL)u.o)
12                     & 0xffffffff);
13 #  else
14   return (opcode_t) fetch_iv_le((INTVAL)u.o);
15 #  endif
16 #endif
17 }
```

Figure 8: Simple Return in src/packfile/pf_items.c.

Simple Return is the first special case of Simple Conditional Compilation. It captures a nested conditional block at the end of a procedure which controls the procedure's return value (Figure 8). The corresponding blueprint pattern on Figure 6g is slightly more general, as it also matches Simple Return occurrences with only two conditional branches instead of four.

To implement this pattern in advice, one branch has to be chosen as the base code, whereas all other ones should be refactored into advice. This approach breaks the symmetry and correlation between the branches. Hence, understanding the code becomes harder. Just as for Simple Conditional Compilation, the right state should be accessible as join point context. An elegant, robust advice implementation of the Simple Return pattern is hard to achieve.

5.3.2 Simple Declaration

```
1  void* parrot_pic_opcode(PARROT_INTERP, INTVAL op){
2    const int core = interp->run_core;
3  #ifdef HAVE_COMPUTED_GOTO
4    op_lib_t *cg_lib;
5  #endif
6
7    if(core == PARROT_SWITCH_CORE ||
8       core == PARROT_SWITCH_JIT_CORE)
9        return (void*) op;
10 #ifdef HAVE_COMPUTED_GOTO
11   cg_lib = PARROT_CORE_CGP_OPLIB_INIT(1);
12   return ((void**)cg_lib->op_func_table)[op];
13 #else
14   return NULL;
15 #endif
16 }
```

Figure 9: Simple Declaration in src/pic.c.

This second special case of Simple Conditional Compilation corresponds to a conditional block at the end of a procedure, with dependencies on local variables of a block earlier on in the procedure body (Figure 9). This reduces to Simple Conditional Compilation if the variables are only used inside the second block, as is the case for Figure 9. Figure 6h expresses this pattern. The "c" label enforces that the two conditional blocks have the same condition.

The corresponding advice implementation in Figure 10 is an example of a refactoring which introduces run-time overhead. Line 17 has to check for a null pointer at run-time in order to distinguish between the return statements on lines 6 and 8, when the incoming `Op` argument of `parrot_pic_opcode` is not `NULL`. Refactoring of the base application [26] is needed to enable a more efficient advice

```
void* parrot_pic_opcode(PARROT_INTERP, INTVAL op){
  const int core = interp->run_core;

  if(core == PARROT_SWITCH_CORE ||
     core == PARROT_SWITCH_JIT_CORE)
      return (void*) op;

  return NULL;
}

void* computed_goto(INTVAL Op) after Jp
      returning (void** Tmp):
 execution(Jp,``parrot_pic_opcode'')
 && args(Jp,[_,Op])
 && have_computed_goto(_){
   op_lib_t* cg_lib;
   if(Op&&(!(*Tmp))){
     cg_lib = PARROT_CORE_CGP_OPLIB_INIT(1);
     *Tmp= ((void**)cg_lib->op_func_table)[Op];
   }
}
```

Figure 10: Aspect implementation of Figure 9.

refactoring of the Simple Declaration pattern. Figure 9 shows that the fine-grainedness and efficiency of the preprocessor sometimes cannot be matched by alternative implementations.

5.3.3 Scattered Conditional Compilation

```
void Parrot_STM_waitlist_wait(Parrot_Interp interp){
  struct waitlist_thread_data *thr;
  thr = get_thread(interp);
  LOCK(thr->signal_mutex);
#if WAITLIST_DEBUG
  fprintf(stderr, "%p: got lock, waiting...\n",
          interp);
#endif
  while (!thr->signaled_p) {
    pt_thread_wait_with(interp, &thr->signal_mutex);
#if WAITLIST_DEBUG
    fprintf(stderr, "%p: woke up\n", interp);
#endif
  }
  UNLOCK(thr->signal_mutex);
#if WAITLIST_DEBUG
  fprintf(stderr, "%p: done waiting.\n", interp);
#endif
}
```

Figure 11: Scattered Conditional Compilation in src/stm/waitlist.c.

The Scattered Conditional Compilation pattern (Figure 11) expresses that procedures within a file contain conditional blocks with the same condition and highly similar code. These blocks are typically also scattered across multiple files, and hence should be an ideal target for aspects. The blueprint pattern on Figure 6i looks for scattered blocks in one file at a time by requiring that at least two conditional regions with the same condition occur in one file. This is weaker than enforcing that the code inside the conditional regions is similar and that the pattern occurs across files, but in practice is a good approximation. The pattern is applied as one of the last ones to avoid incorrect matches of instances of more specific patterns.

There are a number of issues when trying to refactor Scattered Conditional Compilation into advice. First, the conditional regions are often attached to arbitrary statements, which are not necessarily principled join points [26]. Second, Scattered Conditional Compilation instances are quite heterogeneous (differences in strings, variables, etc.), which complicates advice reuse [6]. Annotations in the base code might be able to cover small changes between blocks, but eventually aspect languages with generic advice are needed [1].

5.3.4 Invasive Conditional Compilation

This pattern is similar to Simple Conditional Compilation, but captures conditional regions which appear in the middle of a procedure body `before`, `after` or `around` a procedure call or global variable access (rounded rectangle on Figure 6j). Invasive Conditional Compilation can be modeled as traditional advice on `call` join points. Invasive Conditional Compilation could be extended to macro expansion join points, if the aspect language supports these.

5.3.5 General Fine-grained Conditional Compilation

In general, the preprocessor facilitates fine-grained (de)selection of statements or even parts of tokens, such as the inclusion of `case`-entries of a conditional `switch` structure, or conditional regions at arbitrary spots within a procedure or data definition. Figure 6k shows a pattern for this that consists of sequences of conditional and unconditional C code. As this pattern is the most general one we have defined, it is the final pattern to apply.

Because statement-level join points [11] are harder to quantify and maintain in a robust fashion than the original conditional compilation, only a subset of the General Fine-grained pattern is worthwhile to implement using aspects. This subset can be increased by exposing additional `call` join points in the base program [26], e.g. by refactoring the statement immediately before or after a block into a procedure. However, as files often contain multiple unique conditional blocks, aspectisation may result into many non-reusable advices. Hence, there is no unique way to decouple the base code from General Fine-grained Conditional Compilation.

5.4 Ad Hoc Patterns

The remaining occurrences of conditional compilation after applying the patterns of Figure 6 are ad hoc patterns like conditional inclusion of procedure declarations and header files, or conditional definitions of macros. Personal experience tells us that these are highly specialised and irregular. As such, there is no robust alternative implementation for them, nor is there a simple pattern to identify them. Fortunately, the second step of our case study shows that their importance in the Parrot VM diminishes over time.

6. HISTORIC ANALYSIS OF PARROT VM

For the second step of the case study, the evolution of the number of source code files of the Parrot VM was calculated using "SLOCCount" [38], and for each pattern the total number of matches was recorded. Figure 12 shows how the number of files more than doubled across the Parrot VM releases, whereas the percentage of source files containing Ad Hoc patterns stays below five percent and even drops to less than one percent. Finding out the actual reasons for this decrease is future work, but it indicates that the eleven patterns of Coarse- and Fine-grained Conditional Compilation explain the conditional compilation usage of more than ninety-nine percent of the source code files. This is promising, as the instances of these patterns can be found automatically with the R3V3RS3 infrastructure. In addition, Figure 12 confirms that the Parrot VM uses conditional compilation in a disciplined way.

Figure 13 plots the average number of occurrences per file of nine of the discussed patterns of conditional compilation. The Simple and Simple Return patterns remain more or less 0.1 and are elided for clarity. The evolution of most patterns fluctuates, with

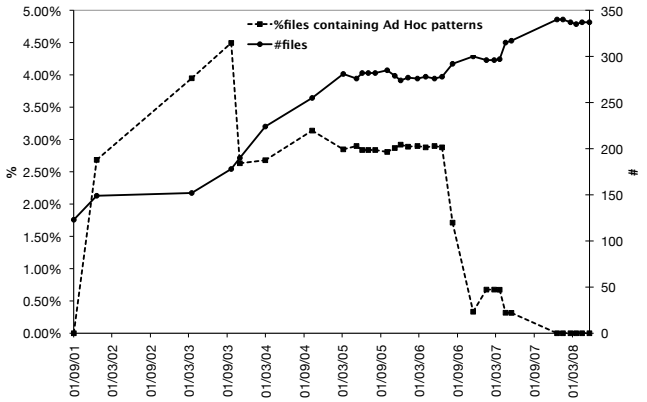

Figure 12: The evolution of the number of files and the percentage of files containing Ad Hoc Patterns for the Parrot VM.

the Invasive and Scattering patterns losing, and Semi-partitioned gaining importance. Conditional Definition is by far the most widespread pattern in the Parrot VM, followed by Multiple Inclusion, General Fine-grained (especially in procedures), Invasive, Partitioned, Simple and Simple Return. The relatively high profile of General Fine-grained (and Multiple Inclusion Protection) is bad news for the refactoring of conditional compilation into aspects, as it is hard to refactor into advice. This acknowledges the intuitive idea that the preprocessor is typically used for tangled concerns rather than scattered ones. Fortunately, Conditional Signature gradually diminishes in importance. Conditional Definition, Simple, Invasive and Partitioned conditional compilation can be refactored into advice, and there are possibilities for General Fine-grained and Simple Return as well, but under certain trade-offs (Table 1).

As we consider the Parrot VM to be an "optimal" case from the perspective of conditional compilation usage, our findings suggest that conditional compilation remains the preferred implementation technique for crosscutting concerns in C/C++ systems, despite its problems. Verifying this claim on other systems first requires finding all occurrences of the conditional compilation patterns summarised in Table 1, followed by an analysis of the evolution of the number of pattern occurrences in a system to determine whether that system would really benefit from a refactoring into advice. Both activities are explicitly supported by the preprocessor blueprint model, as this paper has shown.

7. RELATED WORK

This paper complements recent work on comparing conditional compilation- and aspect-based software product lines [16]. Whereas we consider the potential aspectisation of conditional compilation on its own, i.e. independent from the particular application or purpose of the conditional code, the cited work estimates the benefits of refactoring in the specific context of product line features and constraints. Similar remarks hold for related work in the areas of understanding of preprocessor usage and refactoring of preprocessor code into aspects, which we discuss in the remainder of this section. Note, however, that there are also quite some similarities with the detection of design patterns (e.g. [3]).

7.1 Understanding Preprocessor Usage

Ernst et al. [12] and Krone et al. [22] categorise conditional regions based on the semantics of the preprocessor symbols in their condition. E.g., a region guarded by the `DEBUG` symbol is interpreted as a debugging region. The categorisation of Mennie et al. [27] is not based on the meaning of the preprocessor symbols in a region's condition, but on the semantics of the conditional patterns, like e.g. Multiple Inclusion Protection and Conditional Definition. We instead focus on the syntactical interaction of conditional compilation with the source code, i.e. a region guarded by the `DEBUG` flag is treated differently depending on whether it occurs at the start or at the end of a procedure body. The three ways of categorisation are complementary, however.

Krone et al. [22] visualise for each code region all preprocessor flags which affect it. They can e.g. determine the coupling of build configurations by deducing which flags are implied by others. Unfortunately, the visualisation is overloaded, and cannot be queried. Second, the link between a preprocessor flag and its negation are not conserved, nor the sequential order of conditional regions.

In general, tool support for understanding and refactoring preprocessor usage faces important technical hurdles [5, 17, 23, 25, 33, 37]. A preprocessor-aware parser [13, 36] is needed as well as a way to determine the conditions under which code regions are active [24]. R3V3RS3 takes these challenges into account.

7.2 Refactoring into Aspects

The second category of related work concerns the refactoring of preprocessor usage into aspects. Bruntink et al. [6, 7] and Lohmann et al. [26] refactor preprocessor-aware code into aspects. Inconsistent macro usage precludes automatic refactoring and requires considerably generic advice [6]. Lohmann et al. [26] have found that preparatory refactoring of the base code and/or support for statement-level join points [11] are needed to ensure the presence of the right join points. The C-CLR environment [31] supports refactoring by hiding unused conditional code for one build configuration and by mining for aspects using clone detection techniques. The latter only takes into account the current build configuration.

Various researchers [2, 26, 30, 31] have investigated the refactoring of preprocessor code, especially conditional compilation, into aspects. Previously [2], we have sketched a two-phase process for refactoring into aspects. This approach depends on the outcome of four trade-offs. First, reuse opportunities for the refactored advice depend on the preprocessor's favour of tangled, one-off adaptations ("heterogeneous concerns" [9]). Second, the preprocessor's fine-grainedness asks for statement-level join points [11, 26], but these lead to fragile pointcuts. Third, the ease of aspect refactoring is determined by the developers' discipline regarding preprocessor usage. Fourth, the preprocessor has desirable properties that are not shared by aspects, such as the absence of run-time penalties and weaving logic in the resulting binary.

The fourth trade-off has become less critical because of advances in aspect weaver optimisations. The first three trade-offs are a different matter. To determine their impact on the viability of the proposed two-phase mining process [2], the syntactical interaction between the preprocessor and the source code has to be analysed, which is the focus of this paper. Preprocessor blueprints provide support to reason about these three trade-offs for aspect mining.

Independently from our work, Reynolds et al. [30] have refactored Linux kernel extensions into aspects. They have found that 45% of the configuration-dependent code in procedure definitions and 52% of the conditional fields in data structures are easy to model using `before`/`after`-advice or field introductions respectively (Simple Conditional Compilation). Nearly 10% of all procedures are Conditional Definitions. Reynolds et al. acknowledge the need to prepare the source code to expose the right join points [26].

The work of Reynolds et al. [30] differs from our work in several ways. First, preprocessor blueprints form an explicit model of conditional compilation usage which can be queried for any declaratively specified pattern. Reynolds et al. on the other hand have

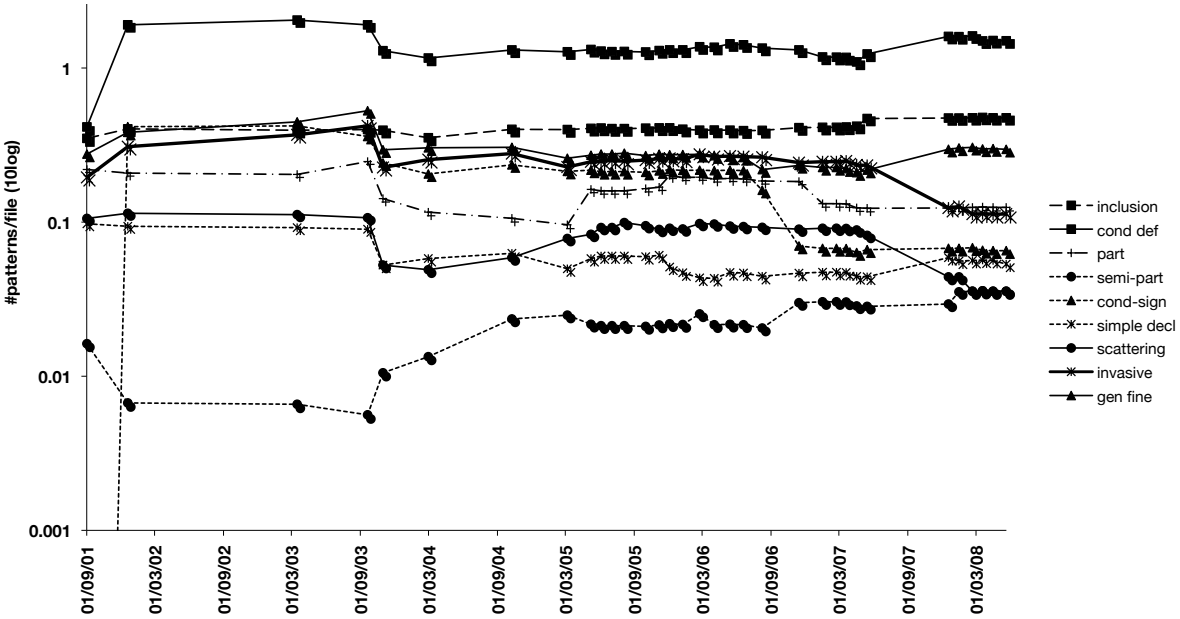

Figure 13: Overview of the average number of pattern occurrences per file across all nesting levels for the analysed releases of Parrot VM. Simple and Simple Return are not shown, as they remain 0.1 throughout.

hard-coded their patterns. Second, our pattern matching approach has enabled us to find occurrences of more complex patterns. Third, we have studied a different subject system (VM instead of operating system), with other development guidelines in place. Our approach confirms the findings of Reynolds et al. [30], and enhances them with additional patterns of conditional compilation usage and a historic analysis of the patterns' popularity.

8. FUTURE WORK

We are applying R3V3RS3 to other systems to further validate the identified patterns of interaction between the preprocessor and the source code, and to detect additional patterns. Despite R3V3RS3, this is time-consuming because one needs to find a system with a good development history and study the evolution of the number of pattern occurrences across its development history. The Linux system is one of our targets, as this enables us to validate the patterns suggested by Figure 12 and to compare our results to those of Reynolds et al. [30]. Finally, we are investigating in more detail why the percentage of Ad Hoc Patterns in the Parrot VM drops significantly, and what kind of patterns are typically temporary.

9. CONCLUSION

This paper has presented three requirements for a model of the syntactical interaction of source code and conditional compilation, and a realisation of these requirements: preprocessor blueprints. This is an abstract model of the nesting and ordering between program statements which correspond to join point shadows or opaque (uninteresting) pieces of code, and it can be queried via a declarative pattern facility. These patterns can express the four commonly known patterns of conditional compilation usage, as well as seven additional patterns we have discovered in the Parrot VM. For each of the patterns, the potential for refactoring into advice is discussed.

By analysing the evolution of the number of occurrences of each pattern in the Parrot VM, we were able to conclude that the eleven patterns capture conditional compilation usage in up to ninety-nine percent of the source files of Parrot VM, and that two of the six most popular patterns are hard to refactor into advice. As the Parrot VM can be considered as a best case scenario from the perspective of refactoring into advice and given the trade-offs derived for most of the discussed aspect implementations, conditional compilation often still is the preferred implementation technique to manage variability in C/C++ systems, despite the tangling and scattering.

Acknowledgements

We are grateful to Yvonne Coady, Celina Gibbs and Chris Matthews for giving us the inspiration for this paper, and to Bart Van Rompaey and Bart Du Bois for the excellent Fetch support. Bram Adams is supported by a BOF grant from Ghent University.

10. REFERENCES

[1] B. Adams. *Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems*. PhD thesis, Ghent University, Ghent, Belgium, March 2008.

[2] B. Adams, B. V. Rompaey, C. Gibbs, Y. Coady, and H. Tromp. Aspect mining in the presence of the C preprocessor. In *Proc. of the 4th Linking Aspect Technology and Evolution Workshop (LATE), AOSD*, Brussels, Belgium, 2008.

[3] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, page 166, Washington, DC, USA, 2001. IEEE Computer Society.

[4] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Softw. Pract. Exper.*, 30(8):907–924, 2000.

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

[6] 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.

[7] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 37–46, Budapest, Hungary, 2005. IEEE Computer Society.

[8] cloc. http://cloc.sourceforge.net.

[9] 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.

[10] B. Du Bois, B. Van Rompaey, K. Meijfroidt, and E. Suijs. Supporting reengineering scenarios with FETCH: an experience report. In *Proc. of the 3rd International ERCIM Workshop on Software Evolution, ICSM*, pages 69–82, Paris, France, October 2007.

[11] M. Eaddy and A. V. Aho. Statement annotations for fine-grained advising. In *Proc. of the Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE), ECOOP*, pages 89–99, Nantes, France, July 2006. Fakultät für Informatik, Universität Magdeburg.

[12] 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.

[13] J.-M. Favre. Preprocessors from an abstract point of view. In *Proc. of the 3rd Working Conference on Reverse Engineering (WCRE)*, page 287, Monterey, CA, USA, 1996. IEEE Computer Society.

[14] J.-M. Favre. Understanding-in-the-large. *Proc. of the 5th International Workshop on Program Comprehension (IWPC)*, pages 29–38, March 1997.

[15] Fetch. http://lore.cmi.ua.ac.be/fetchWiki/index.php.

[16] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proc. of the 30th international conference on Software engineering (ICSE)*, pages 261–270, Leipzig, Germany, 2008. ACM.

[17] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, Univ. of Illinois, 2005.

[18] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. of the 3rd international conference on Aspect-oriented software development (AOSD)*, pages 26–35, Lancaster, UK, 2004. ACM.

[19] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[21] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In *Proc. of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE), ECOOP*. Springer Verlag, Oslo, Norway, June 2004.

[22] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. of the 16th international conference on Software engineering (ICSE)*, pages 49–57, Sorrento, Italy, 1994. IEEE Computer Society.

[23] B. Kullbach and V. Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE)*, page 3, Stuttgart, Germany, 2001. IEEE Computer Society.

[24] M. Latendresse. Rewrite systems for symbolic evaluation of C-like preprocessing. In *Proc. of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR)*, page 165, Tampere, Finland, 2004. IEEE Computer Society.

[25] P. Livadas and D. Small. Understanding code containing preprocessor constructs. *Proc. of the IEEE 3d Workshop on Program Comprehension*, pages 89–97, November 1994.

[26] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. *SIGOPS Oper. Syst. Rev.*, 40(4):191–204, 2006.

[27] 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.

[28] Parrot VM. `http://parrotcode.org/`.

[29] Parrot VM design documents. `http://www.parrotcode.org/docs/pdd/`.

[30] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the feasibility of an AOSD approach to Linux kernel extensions. In *Proc. of the 7th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, Brussels, Belgium, 2008.

[31] 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.

[32] 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, Baltimore, MD, USA, June 1992. USENIX Association.

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

[34] A. Sutton and J. Maletic. How we manage portability and configuration with the C preprocessor. In *Proc. of the 23rd International Conference on Software Maintenance (ICSM)*, Paris, France, October 2007.

[35] Uncrustify. http://uncrustify.sourceforge.net/.

[36] 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.

[37] 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.

[38] D. A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/.