R3V3RS3: Querying for Syntactical Patterns of Conditional Compilation Usage

Tabletop demonstration proposal

Bram Adams*, Ahmed E. Hassan
SAIL, School of Computing, Queen's University
156 Barrie street
K7L 3N6, Kingston (Canada)
+1 613 533 6802
{bram,ahmed}@cs.queensu.ca

Herman Tromp
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41
B-9000, Ghent (Belgium)
herman.tromp@ugent.be

Wolfgang De Meuter
PROG, DINF, Vrije Universiteit Brussel
Pleinlaan 2
B-1050, Brussels (Belgium)
wdmeuter@vub.ac.be

ABSTRACT

Conditional compilation is a conditional text inclusion facility which has been used for ages in C/C++ code as the primary mechanism for implementing crosscutting concerns like debugging, configuration, etc. In order to compare the benefits and drawbacks of using conditional compilation for this with those of more modern aspect-oriented techniques, it is important to find out what the typical patterns of syntactical interaction between conditional regions and normal source code are. R3V3RS3 is a reverse-engineering tool which reifies the "preprocessor blueprint model" of a C system, i.e. a model of the syntactical interaction between source code and conditional compilation regions. This model can be queried for interesting patterns which have been specified declaratively. To cope with the text processing nature of conditional compilation, R3V3RS3 is built on top of Fetch, a robust C/C++ analysis tool chain. R3V3RS3 has been applied to the Parrot VM to discover syntactical patterns of conditional compilation usage and to study the evolution of the number of pattern occurrences. In the demo, we apply the identified patterns of conditional compilation usage to a smaller VM and use these findings to decide which conditional compilation regions are interesting to refactor into aspects and which ones are not.

1. PROBLEMS ADDRESSED

The conditional compilation facility in C/C++ conditionally includes program text for compilation based on a logical condition expressed in terms of preprocessor flags. These are degenerate macros which correspond to a constant or which are only known to be (un)defined. Figure 1 shows a C code example with a conditional region on lines 2–4. If the `THREAD_DEBUG` preprocessor flag is defined, the preprocessor retains the code on line 3 for compilation and removes lines 2 and 4. Otherwise, lines 2–4 all disappear. Conditional compilation provides a fine-grained mechanism to control variability points in C/C++ source code [1, 18].

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

Figure 1: Simple Conditional Compilation in src/thread.c.

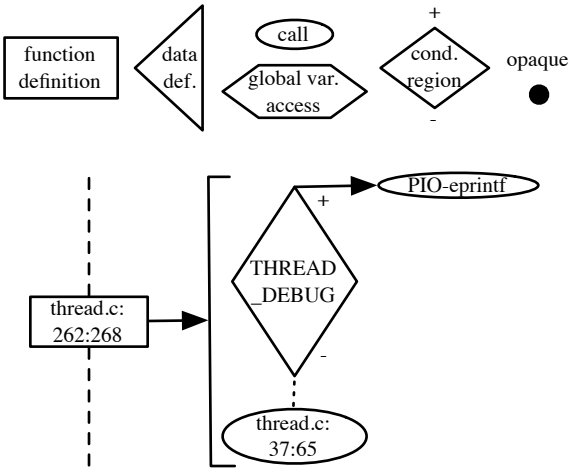
There seems to be a clear link between conditional compilation and aspects. The `THREAD_DEBUG` condition on line 2 of Figure 1 e.g. can be interpreted as a pointcut, the conditional code on line 3 as advice, and the physical position of the conditional region (i.e. the start of the `pt_transfer_sub` procedure body) as the join point shadow[1] [11] at which the advice should match. The major difference with aspects is that a conditional region's condition is statically evaluated by the C preprocessor, whereas aspects conceptually are woven at run-time. Only if aspect weavers statically exploit the known preprocessor flags, aspects can be as efficient as conditional compilation [1].

The problem one is confronted with when studying the relation between conditional compilation and aspects, is that conditional compilation is heavily intertwined with the C code and even is allowed to violate the C syntax rules. On average 4% of all lines of source code contains conditional compilation directives and 37% of the source code is actually controlled by conditional compilation [6]. Hence, a considerable part of the source code is mixed with conditional constructs, and as such hard to understand and analyse. In addition, the preprocessed program can be one of a multitude of C programs depending on which flags have been (un)defined. As a consequence, the C preprocessor is an important source of errors and of confusion [8]. Hence, tool support is needed to really understand how the C preprocessor, especially conditional compilation, interacts with the normal C source code.

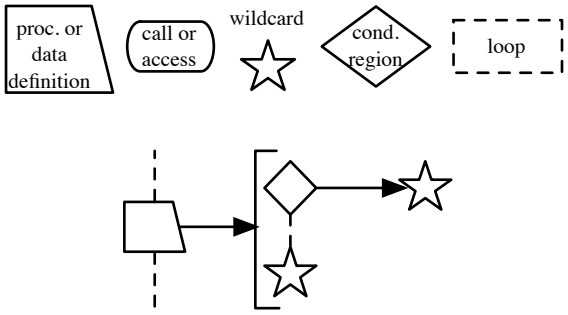
More in particular, we are interested in how conditional compilation syntactically interacts with the source code, i.e. how code portions are nested within a conditional region, what the relative position of regions inside a function definition is, how conditional

*Both contact person and presenter.

[1]For each run-time join point (e.g. a `call` join point), a corresponding construct in the source code can be found which is executed by the join point at run-time (e.g. the actual procedure call). This is exploited by compile-time weavers to statically weave aspects.



(a) Blueprint model for Figure 1.



(b) Blueprint pattern for Figure 1.

definitions are nested, etc. As conditional compilation usage can be very complex, it is important to have the ability to identify and retrieve all occurrences of important patterns of conditional compilation, such as the example pattern in Figure 1. This is exactly the functionality which is offered by R3V3RS3[2], our reverse-engineering tool for conditional compilation usage. R3V3RS3 has enabled us to study the existing patterns of conditional compilation usage in the Parrot VM and to analyse the evolution of pattern usage throughout the lifetime of this system. These results are discussed in a companion paper [2]. This demo proposal focuses on the R3V3RS3 tool itself.

First, we focus on the relevance of R3V3RS3 to AOSD (Section 2). Then, we discuss the design of the tool, which is centered around the "preprocessor blueprint model" (Section 3). The mapping of this design on an actual implementation is explained in Section 4, which is followed by a brief account of related tools and techniques (Section 5). Finally, Section 6 gives an outline of the proposed tabletop demonstration.

2. RELEVANCE TO AOSD

Many researchers [2, 14, 16, 17] propose to replace conditional compilation with aspects, because a conditional region typically corresponds to a scattered and tangled implementation of a crosscutting concern like debugging support, tracing, platform-dependent logic, etc. Refactoring a conditional region into explicit advice with the region's condition as the pointcut promises a more semantic description of the intent of the conditional regions, and would be easier to maintain and understand, as it makes the base code independent from the C preprocessor and the build system. The conditional region in Figure 1 could e.g. be refactored into `before`-advice on the procedure's execution join point. However, it is important to find out which patterns of conditional compilation usage could benefit from refactoring into aspects, and whether aspect languages can express these patterns. As argued in Section 1, the interaction between conditional compilation and the source code is very complex. Tool assistance is needed, and this is what R3V3RS3 provides.

The high-level goal of our research is actually to grasp the role played by the combination of C preprocessor and build system [1] in software development, and how this relates to more recent technologies like aspect-orientation. Instead of using inter-type declaration[3] (ITD), C systems either use a combination of conditional compilation and file inclusion, or rely on the build system to add or withdraw a compilation module to/from the build. Understanding the benefits and drawbacks of these time-proven techniques can shine new light on the fundamental concepts behind aspect-orientation.

3. UNIQUENESS OF DESIGN AND IMPLEMENTATION

R3V3RS3 implements the "preprocessor blueprint model" [2]. This is a graphical model which satisfies three requirements which were derived from the four syntactical patterns of conditional compilation which are commonly found in literature [2, 6, 15, 16]:

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.

This section explains the conceptual idea behind the preprocessor blueprint model. Our explanation is based on Figure 2a and Figure 2b, which corresponding to the blueprint model and a blueprint pattern of the example in Figure 1. The full details on the preprocessor blueprint model can be found elsewhere [2].

3.1 Focus on Interesting Statements

Depending on the join point model of the aspect language which is used, some program statements are more interesting than others. All aspect languages [1] for C/C++ fancy join points of which the shadow corresponds to a procedure or type definition, file (for ITD of procedures and data), procedure call or global variable reference. Assignments, pointer dereferences and macro expansions are not interesting, as they are not directly supported by most as-

[2] Pronounced as "reverse".

[3] Aspect-oriented technique to add fields and methods to data structures or classes.

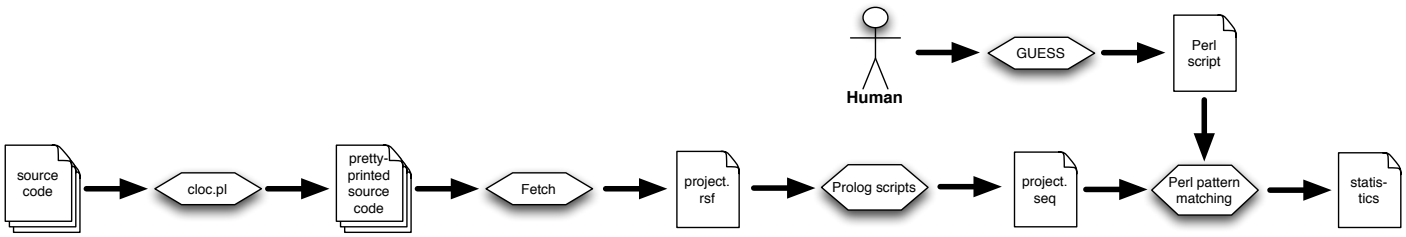


Figure 2: High-level architecture of R3V3RS3.

pect languages for C/C++, and in general it is hard to define robust pointcuts for these fine-grained join points. Still, they must not be dismissed completely, because their presence relative to the interesting statements is one of the major problems when trying to come up with a robust pointcut for the latter. Hence, the preprocessor blueprint model coalesces uninteresting statements and treats them as "opaque" sequences of statements by just recording their presence and position relative to the interesting statements.

Figure 2a is the preprocessor blueprint model of Figure 1. As the latter only contains a function definition, two procedure calls and a conditional region, there are no opaque statements. Each type of statements has its own kind of node. A conditional node is labeled with the conditional expression, whereas definition and call nodes are labeled with the file name and line numbers of the called or defined procedure, or the file name if a library function is called.

3.2 Nesting and Ordering of Model Elements

The second requirement emphasises nesting and ordering as the primary relations between the model's elements. First, a conditional region is either enclosed completely by another conditional region, or not at all ("inclusion dependence" [6]). Second, to reason about patterns of conditional compilation usage, the ordering of the model elements relative to each other is crucial. It does matter e.g. 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 are represented both.

On Figure 2a, the `pt_transfer_sub` procedure contains a sequence of a conditional region and a procedure call. The conditional region only has a positive branch, which contains a procedure call. A preprocessor blueprint models nesting as left-to-right edges, and ordering as top-down edges.

3.3 Declarative Pattern Matching Facility

To retrieve all occurrences of a pattern of conditional compilation, a declarative pattern matching facility is needed. Most tools hard-code a set of patterns [16], or only provide an imperative way to specify and detect occurrences of patterns. PCp^3 [3, 6] e.g. is an extensible C preprocessor based on callbacks. Occurrences of individual preprocessor constructs are easy to handle, but detection of sequences of events requires to manually implement a state machine. A declarative approach is able to circumvent this limitation.

Figure 2b shows a query to find all occurrences of a preprocessor blueprint like Figure 2a, i.e. a conditional region at the beginning of a procedure definition. This is modeled as the nesting inside a procedure definition of a conditional region and sequencing this region before a random block of statements (opaque or not). The conditional region can contain any (sequence of) statement(s). The dashed line through the trapezium-shaped node indicates that this query is in relative mode, i.e. it can match at any level of nesting inside a preprocessor blueprint. A query in absolute mode has to match with the blueprint model at the highest level of nesting (file-level). More details can be found elsewhere [2].

4. UNDERLYING IMPLEMENTATION TECHNIQUES AND TECHNOLOGIES

R3V3RS3 is a prototype implementation of the preprocessor blueprint model and pattern matching facility discussed in the previous section. Figure 2 shows its architecture, which is based on a robust C parser and Perl regular expression matching. In order to improve the results of later phases, the source code first is pretty-printed using uncrustify [19] and stripped from comments and blank lines by cloc [4]. The pretty-printed version is then parsed by Fetch [9]. This is a reverse-engineering tool chain based on robust C/C++ fact extraction and a lightweight parser [5]. Its output is a graph in the Rigi Standard Format (RSF) that describes the program AST and preprocessor usage. We use Prolog scripts to generate a textual representation of preprocessor blueprints from the RSF file.

A simple GUI front end enables developers to compose a blueprint pattern query. It is based on the graph manipulation environment GUESS [10]. Python scripts on top of GUESS transform the query into a combination of Perl regular expressions in order to achieve the semantics outlined in Section 3.3. R3V3RS3 finds all occurrences of a pattern in a preprocessor blueprint (taking into account relative/absolute mode), and removes these patterns from the blueprint if desired. R3V3RS3 gives as output the number of occurrences of each pattern as well as the filtered preprocessor blueprint.

5. RELATED TOOLS

Related work is discussed in detail in [2]. Here, we focus on related tools in the areas of understanding of preprocessor usage and refactoring of preprocessor code into aspects.

In general, tools for understanding and refactoring preprocessor usage require a preprocessor-aware parser [7] and a means to determine the conditions under which conditional regions are active [13]. R3V3RS3 uses Fetch [5] for these. Ernst et al. [6] and Krone et al. [12] categorise conditional compilation regions based on the semantics of the preprocessor symbols in their condition instead of on the syntactical interaction with the source code. Ernst et al.'s Pcp^3 tool for this has been discussed in Section 3.3. Krone et al. visualise for each conditional region all preprocessor symbols which affect it in a lattice. This allows e.g. to determine the coupling of build configurations by deducing which flags are implied by others. Unfortunately, the visualisation contains too much information, and cannot be queried. In addition, the sequential order of conditional regions is not conserved. Finally, Mennie et al. [15] have the infrastructure to declaratively match patterns in a fact base representation of the source code, but they do not use it to find the occurrences of patterns of conditional compilation usage. 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 search for more complex interactions.

The second category of related work concerns the refactoring of preprocessor usage into aspects. The C-CLR environment [17] hides unused conditional code for a given build configuration and mines for aspects in the conditionally selected code. Independently from our work, Reynolds et al. [16] have refactored Linux kernel extensions into aspects. Their work differs from our approach [2] in several ways. First, preprocessor blueprints form an explicit and queryable model of conditional compilation usage. Reynolds et al. on the other hand have hard-coded their patterns. Second, our declarative approach enables us to find additional patterns. Third, although we have studied a different subject system, our approach confirms the findings of Reynolds et al. [16], and enhances them with additional patterns of conditional compilation usage and a historic analysis of the patterns' popularity.

6. DESCRIPTION OF DEMO

The goal of the demo is to show the workflow of using R3V3RS3 for retrieving instances of interesting patterns of conditional compilation usage by applying R3V3RS3 on the CSOM VM, i.e. a small VM for a Smalltalk dialect. It is a port to C of the SOM VM from the University of Århus, done by people at the Hasso-Plattner-Institut, where CSOM is used in courses on VMs. The demo provides a good opportunity to verify whether or not the conditional compilation patterns we have identified in the larger Parrot VM [2] are representative for CSOM as well.

The outline of the proposed demo looks like this:

1. Very short recap about the goals and architecture of R3V3RS3, and the CSOM VM.
2. The textual blueprint model of CSOM is generated and highlighted (Figure 3).
3. We use R3V3RS3's GUI front end to compose a conditional compilation pattern like the one on Figure 2b (Figure 4).
4. This pattern is transformed into a Perl script, which is then briefly discussed to show why the preprocessor blueprint patterns are expressed in a graphical formalism and not as (complex) regular expressions (Figure 5).
5. The CSOM blueprint model is queried for the pattern and the resulting matches are discussed (Figure 6).
6. Finally, we apply all patterns identified for the Parrot VM [2] and compare the distribution of the number of pattern occurrences to the one we found for the Parrot VM (Figure 7). This enables us to discuss potential refactoring of the identified conditional compilation patterns into aspects.

Acknowledgements

We are grateful to Yvonne Coady, Celina Gibbs and Chris Matthews for giving us the inspiration for this work, and to Bart Van Rompaey and Bart Du Bois for the excellent Fetch support.

7. 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, H. Tromp, W. D. Meuter, and A. E. Hassan. Can we refactor conditional compilation into aspects? In A. Moreira and C. Schwanninger, editors, *Proc. of the 8th International Conference on Aspect-Oriented Software Development (AOSD)*, Charlottesville, Virginia, USA, March 2009. To appear.

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] GUESS. http://graphexploration.cond.org/.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[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] 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.

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

Figure 3: Generated textual representation of preprocessor blueprint model.

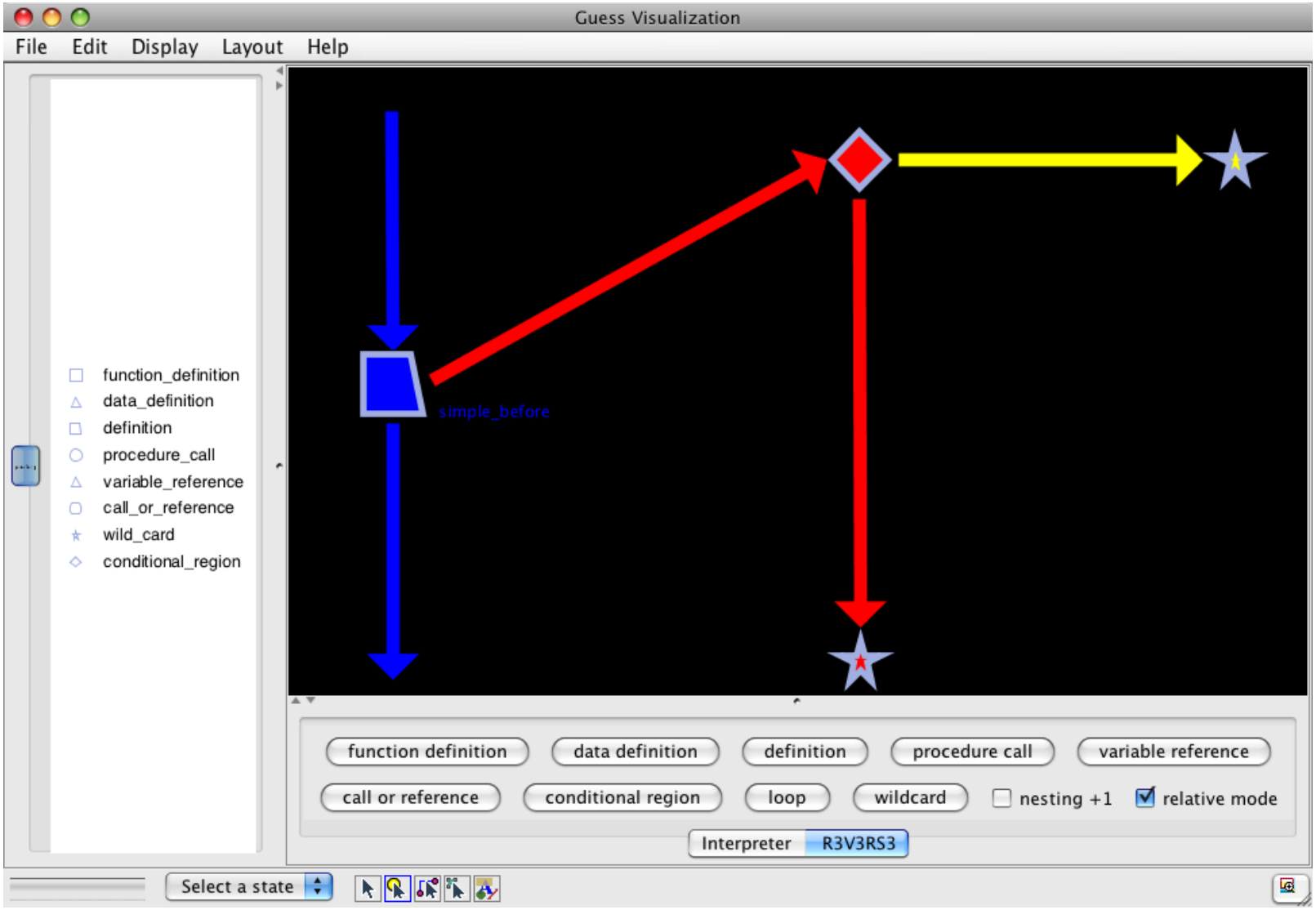


Figure 4: R3V3RS3 GUI for specifying patterns.

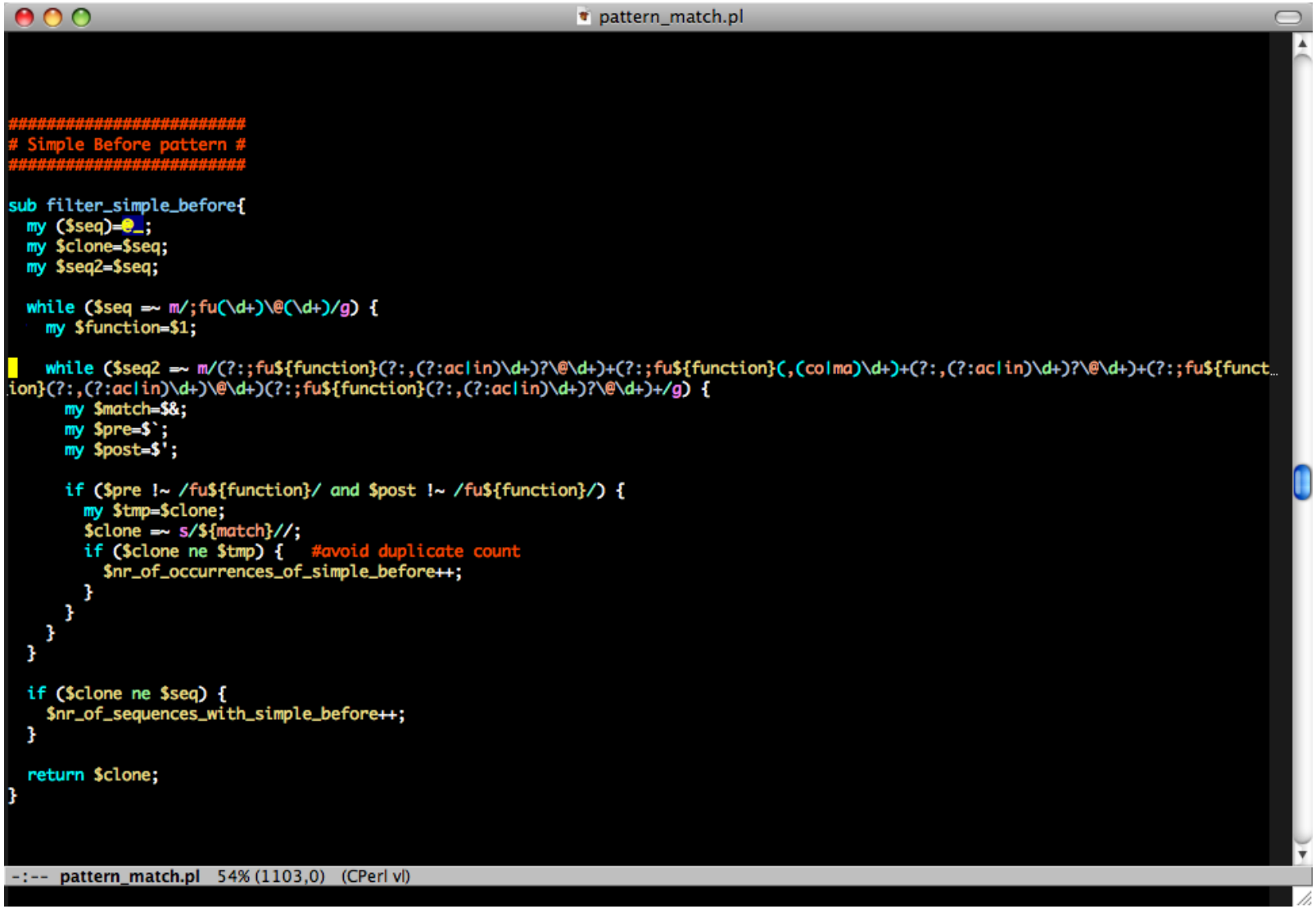


Figure 5: Generated Perl code for pattern.

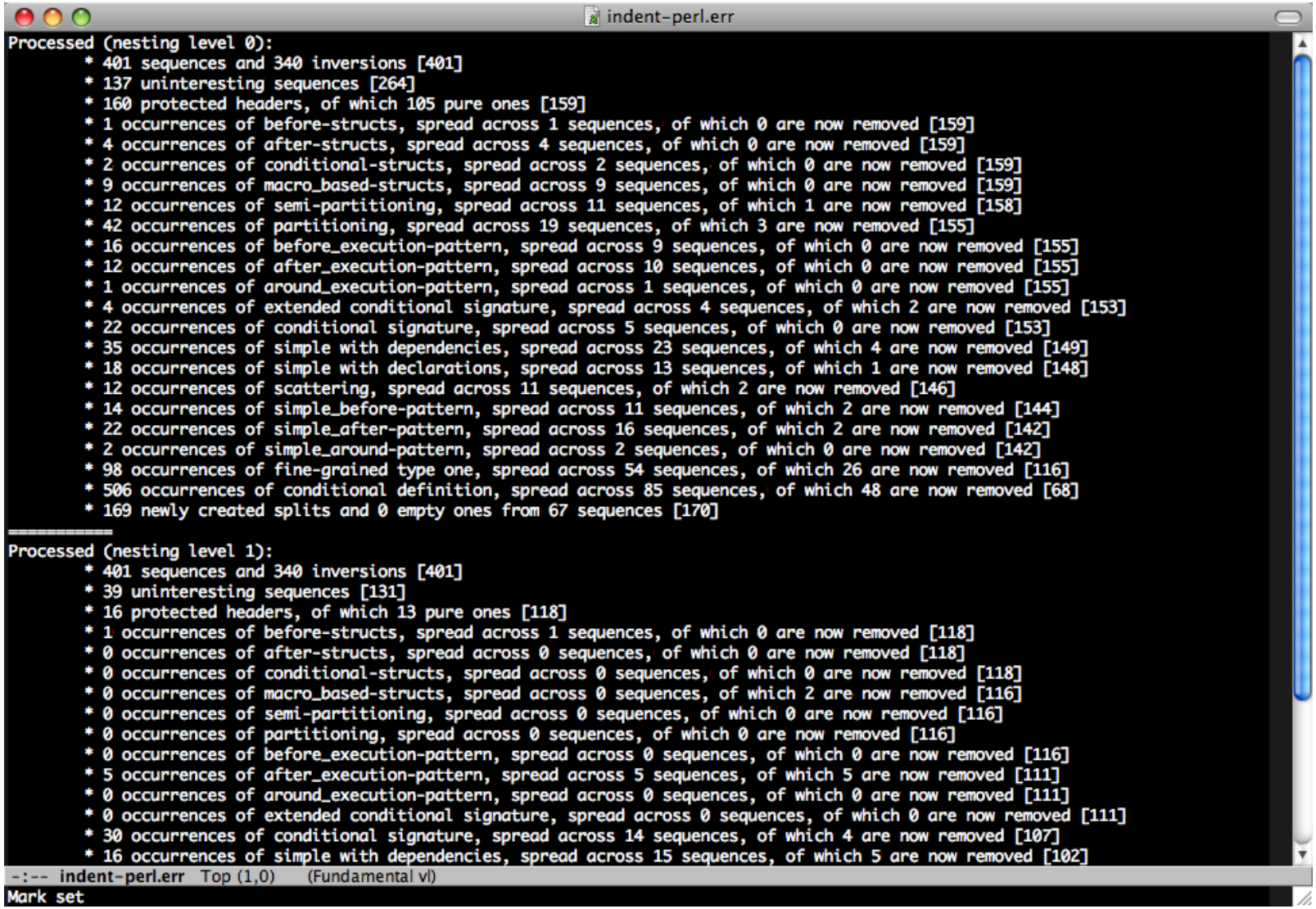


Figure 6: Statistics for pattern matching.

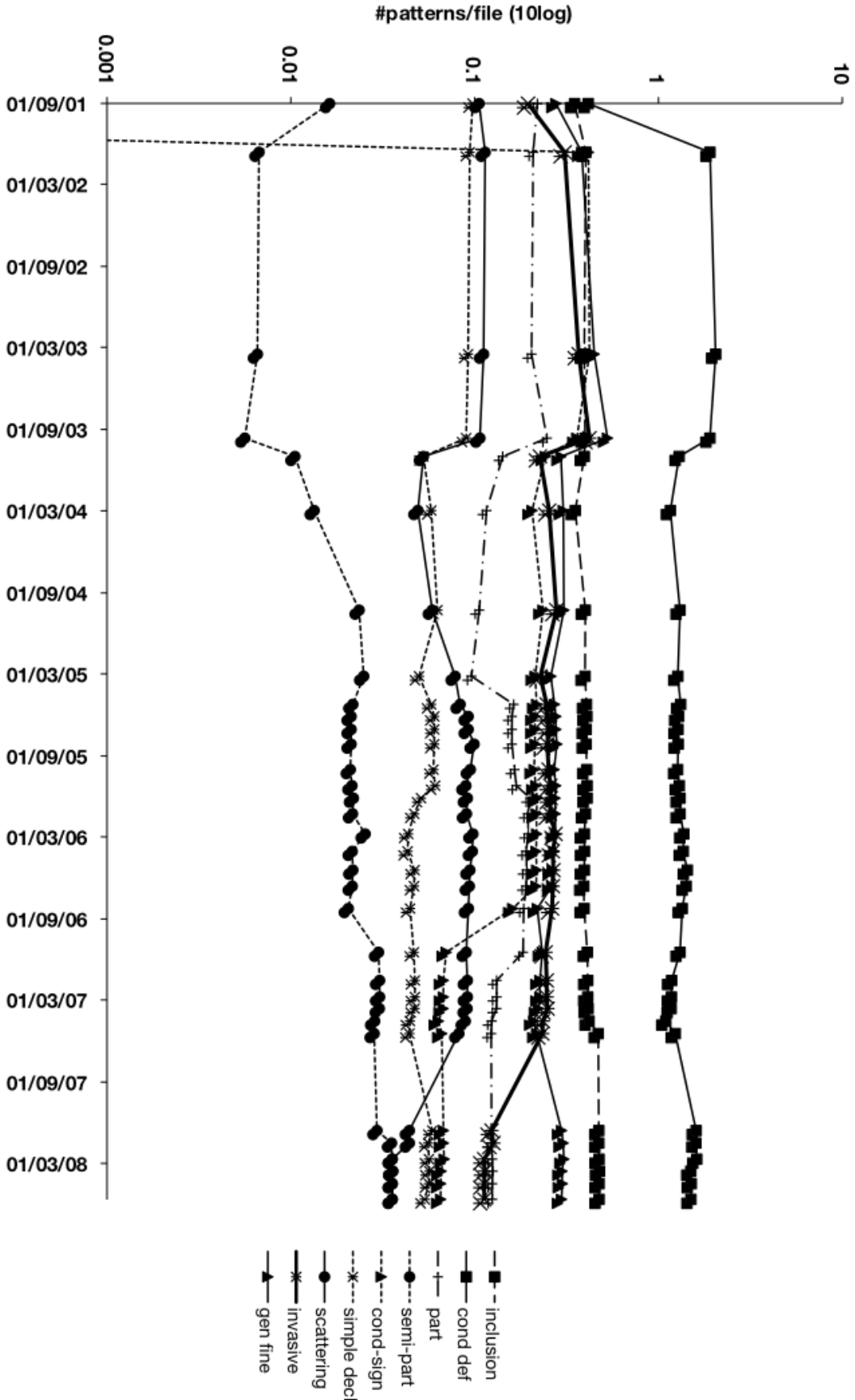


Figure 7: Excel graph with distribution of number of pattern occurrences.