Discipline Matters: Refactoring of Preprocessor Directives in the `#ifdef` Hell

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner,
Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca

Abstract—The C preprocessor is used in many C projects to support variability and portability. However, researchers and practitioners criticize the C preprocessor because of its negative effect on code understanding and maintainability and its error proneness. More importantly, the use of the preprocessor hinders the development of tool support that is standard in other languages, such as automated refactoring. Developers aggravate these problems when using the preprocessor in undisciplined ways (e.g., conditional blocks that do not align with the syntactic structure of the code). In this article, we proposed a catalogue of refactorings and we evaluated the number of application possibilities of the refactorings in practice, the opinion of developers about the usefulness of the refactorings, and whether the refactorings preserve behavior. Overall, we found 5670 application possibilities for the refactorings in 63 real-world C projects. In addition, we performed an online survey among 246 developers, and we submitted 28 patches to convert undisciplined directives into disciplined ones. According to our results, 63% of developers prefer to use the refactored (i.e., disciplined) version of the code instead of the original code with undisciplined preprocessor usage. To verify that the refactorings are indeed behavior preserving, we applied them to more than 36 thousand programs generated automatically using a model of a subset of the C language, running the same test cases in the original and refactored programs. Furthermore, we applied the refactorings to three real-world projects: *BusyBox*, *OpenSSL*, and *SQLite*. This way, we detected and fixed a few behavioral changes, 62% caused by unspecified behavior in the C programming language.

Index Terms—Configurable Systems, Preprocessors, and Refactoring

1 Introduction

The C preprocessor is a language-independent tool for lightweight meta-programming that provides no perceptible form of modularity [44]. The preprocessor is used in many projects written in C. Developers use preprocessor directives, such as `#ifdef` and `#endif`, to mark blocks of source code as optional or conditional, with the purpose of tailoring software systems to different hardware platforms, operating systems, and application scenarios.

Researchers and practitioners have been criticizing the C preprocessor because of its negative effect on code understanding and maintainability and its error proneness [10], [5], [26], [31], [30], [33]. More importantly, the preprocessor hinders the development of tool support that is standard in other languages, such as automated refactoring [49], [23], [15], [29], [21], [48]. Developers aggravate these problems when using the preprocessor in undisciplined ways, for example, using preprocessor directives that split up a statement or expression [10], [5], [14], [26]. We call preprocessor usage that does not respect the syntactic structure of the source code *undisciplined* (e.g., wrapping a single bracket without its corresponding closing one).

To resolve undisciplined preprocessor usage, in a previous work, we proposed a catalog of refactorings (and we developed an *Eclipse* plug-in to automate the refactorings) [32]. The catalog contains 14 refactorings to resolve undisciplined directives divided into 4 categories: single statements, conditions, wrappers, and comma-separated elements. In this article, we report on a mixed-method evaluation [8], [9] of our refactorings to answer the following research questions:

- **RQ1:** What is the number of possibilities to apply the refactorings in practice?
- **RQ2:** What opinion do developers have on our catalog of refactorings in practice?
- **RQ3:** Do the refactorings of the catalog preserve program behavior?

To answer **RQ1** (number of possibilities), we analyzed 63 open source C projects searching for opportunities to apply the refactorings in practice. We considered C projects of different sizes and from various domains, such as games, operating systems, Web servers, and database systems. Overall, we found 5670 application possibilities, in 59 out of the 63 open-source projects (one refactoring possibility for every 3704 lines of code), showing that our refactorings are indeed relevant to real-world C projects.

- *F. Medeiros is affiliated with the Federal Institute of Alagoas, AL, 57020-600, and with the Department of Computing and Systems, Federal University of Campina Grande, PB, 58429-900, Brazil. R. Gheyi is affiliated with the Department of Computing and Systems, Federal University of Campina Grande, PB, 58429-900, Brazil. M. Ribeiro, B. Ferreira, Luiz Carvalho, and B. Fonseca are affiliated with the Computing Institute, Federal University of Alagoas, Maceió, AL, 57072-900, Brazil. C. Kästner is affiliated with the Institute for Software Research, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA. S. Apel is affiliated with the Department of Informatics and Mathematics, University of Passau, 94032, Germany.*

We answer **RQ2** (opinion of developers) with two different evaluations. First, we analyzed data from an online survey among 246 developers. We asked them about their preferences showing pairs of behaviorally equivalent code snippets: (1) the original code from a real-world project with undisciplined preprocessor usage, and (2) a disciplined version of the original code snippet, created by applying one of our refactorings. Second, we selected a subset of application possibilities in the studied subject systems and submitted patches to the respective project converting undisciplined directives into disciplined ones to get feedback from developers. The results of our survey reveal that 63% of developers prefer to use the refactored code (i.e., the disciplined version of the code) instead of using the preprocessor in undisciplined ways. Likewise, we received positive feedback from developers when submitting patches to resolve undisciplined directives.

To answer **RQ3** (behavior preservation), we created a model of a subset of the C language and a corresponding code generator, based on *Alloy* [19], to automatically generate programs with application possibilities for our refactorings, along with corresponding test cases. In addition, we applied our refactorings in *BusyBox*, *OpenSSL*, and *SQLite*, three real-world projects with test cases available. For all subjects (Alloy-based and real-world), we ran the test cases before and after applying our refactorings to check behavior preservation. We found and fixed some behavioral changes in one refactoring of our catalog, and a few behavioral changes in the refactoring implementations. Most behavioral changes (62%) are related to unspecified behavior in the C language, though. We modified one refactoring that introduced behavioral changes and updated another refactoring with feedback from developers. We found no behavioral changes when running the test cases before and after applying the refactorings to the three real-world projects.

In summary, the main contributions of this article are:

- We present a mixed-method evaluation of our catalog and implementation of refactorings to remove undisciplined preprocessor directives. The results provide evidence that developers prefer to use the refactored code instead of using the preprocessor in undisciplined ways;
- We discuss a study to understand the opinion of developers regarding our refactorings by submitting patches to 28 C projects. Developers accepted 21 (75%) out of 28 patches that we submitted to the subject systems converting undisciplined directives into disciplined ones;
- We present a study to verify behavior preservation in our refactorings. Our findings increase confidence that our refactorings resolve undisciplined directives without introducing behavioral changes;
- We show a large-scale study to count the number of possibilities to apply our refactorings in 63 real-world C projects. There are 5670 possibilities to use our refactorings in the projects that we study.

This work builds upon our prior work on studying `#ifdef` directives and possible refactorings. Specifically, we interviewed 40 developers and performed a survey with 202 participants in our previous study to learn about the opinion of developers regarding undisciplined directives and its common problems, such as code understanding, maintainability, and error proneness [30]. By studying 12 C real-world projects, we identified patterns of undisciplined preprocessor directives and we proposed a preliminary version of our catalogue of refactorings to remove these patterns of undisciplined directives [32]. In this preliminary version, we evaluated the refactorings showing that they were syntactically corrected, that is, we did not introduce syntax errors with our refactorings. This article significantly expands on prior work by implementing and evaluating the refactorings with multiple studies. We complement our previous studies by submitting patches to remove undisciplined preprocessor directives and by performing a totally new survey with 246 developers. Furthermore, we performed studies to verify behavior preservation in our refactorings using automatically generated programs and three real-world systems. A replication package for our current study is available at the project's *Web* site.[1]

2 Undisciplined Preprocessor Usage

The lexical operation mode, which allows introducing undisciplined directives at arbitrary tokens, is one of the most criticized aspects of the C preprocessor [10], [5], [14], [26], [30], [31], [33]. Undisciplined preprocessor directives do not respect the syntactic structure of the source code (e.g., wrapping a single bracket without its correspondent closing one) [26]. For instance, Figure 1 (a) presents part of the source code of *Vim* including preprocessor directives that we consider as undisciplined. In this case, the #ifdefs wrap only part of expressions. In Figure 1 (b), we present an alternative, disciplined version of the code, in which the preprocessor directives surround entire statements only.

In previous work, we gathered evidence that developers do not recommend using undisciplined directives [30]. By interviewing 40 developers and by performing a survey among 202 participants, we found that most developers agree that undisciplined preprocessor usage influences code understanding (88 %), maintainability (81 %), and error proneness (86 %) negatively [30]. The developers emphasize that they would not use undisciplined directives because they decrease code readability, obfuscate the control flow, and make the code difficult to evolve and maintain. For example, one developer elaborated on the aforementioned problems, saying: *"I avoid this kind of directives; they make the source code hard to understand and maintain. My gut feeling keeps screaming possible bugs when I'm faced with a code like that."* Another developer recommends to discourage or disallow undisciplined directives through code guidelines. The

1. http://fmmspg.appspot.com/refactorings/index.html

guidelines on coding style of the *Linux Kernel*, for example, guide developers to avoid undisciplined directives, saying: "*prefer to compile out entire functions, rather than portions of functions or portions of expressions. Rather than putting an* `#ifdef` *in an expression, factor out part or all of the expression into a separate helper function and apply the conditional to that function.*" An interviewed *Linux Kernel* developer mentioned that code reviewers regularly ask code contributors to rewrite patches to follow the *Linux Kernel* guidelines regarding undisciplined preprocessor usage. The same developer emphasized that there are discussions regarding undisciplined directives in several mailing list posts, showing that many developers care about undisciplined preprocessor directives.

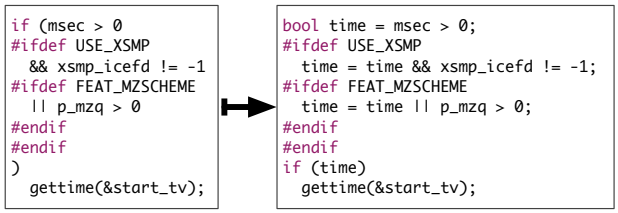


Fig. 1: (a) Code snippet of *Vim* with undisciplined preprocessor directives, and (b) the corresponding disciplined version.

3 Catalog of Refactorings

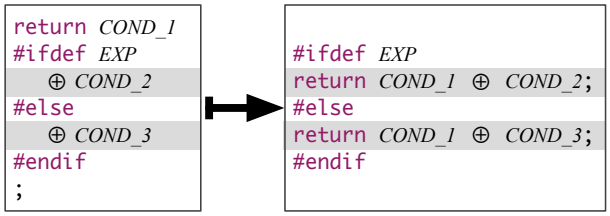
To convert undisciplined directives into disciplined ones, we present our catalog of refactorings in this section. Each refactoring is a unidirectional transformation that consists of two templates of C code snippets: the left-hand side and the right-hand side. The left-hand side defines a template of C code that contains undisciplined preprocessor usage. The right-hand side is a corresponding template for the refactored code without undisciplined preprocessor usage. We can apply a refactoring whenever the left-hand side template is matched by a piece of C code and when it satisfies the preconditions ($\rightarrow$). A matching is an assignment of all meta-variables in the left-hand side/right-hand side templates to concrete values from the source code. We highlight meta-variables using capital letters, and we use the symbol $\oplus$ to represent arbitrary binary operators. Any element not mentioned in both C code snippets remains unchanged, so the refactoring templates only show the differences among pieces of code.

Next, we explain each refactoring of our catalog in detail. We classify our refactorings into four categories: *single statements*, *conditions*, *wrappers*, and *comma-separated elements*.

3.1 Single Statements

In Refactoring 1, we present our refactoring to resolve undisciplined preprocessor usage within single statements.[2] In this refactoring, we duplicate language tokens to encompass with preprocessor directives only entire statements. Notice that we duplicate the token *COND_1* to make the preprocessor directive disciplined. We use a `return` statement as an example, but we handle other statements with subexpressions in the same way.

Refactoring 1: ⟨undisciplined returns⟩



Notice that *COND_1* appears twice in Refactoring 1. To avoid this duplication, our catalogue provides a variation of Refactoring 1, which uses a fresh local variable to keep *COND_1*, as we present in Figure 2. The catalogue also provides variations that use fresh local variables to keep *COND_2* and *COND_3* in situations in which these subexpressions are complex enough to justify the introduction of new variables. In this way, developers can select the variation that best fits their concerns.

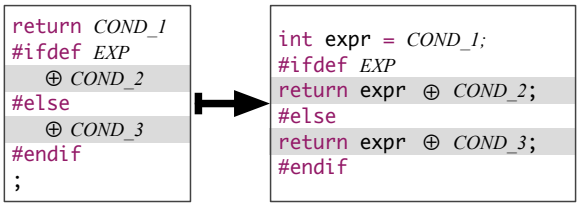


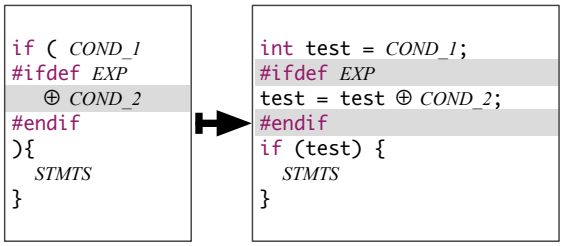
Fig. 2: Variant of Refactoring 1 to deal with complex conditions and to avoid code duplication.

3.2 Conditions

To resolve undisciplined directives surrounding Boolean expressions (used in `if` and `while` statements), we propose Refactoring 2. In this refactoring, we use a fresh variable to maintain the statement's conditions. Specifically, we define a precondition that the code is not using the specific identifier (`test`), as we cannot define variables with the same identifier in the same scope. Notice that the `test` identifier is not suitable for all situations, we use this identifier because the tool needs an identifier for the local variable to apply the refactoring. However, developers might apply a renaming refactoring to choose a suitable identifier based on the real purpose of the source code. We refactor `while` statements with undisciplined conditions similarly.

2. A single statement contains no compound blocks. Examples of single statements are variable initializations, function calls, and return statements.

Refactoring 2: ⟨undisciplined `if` conditions⟩



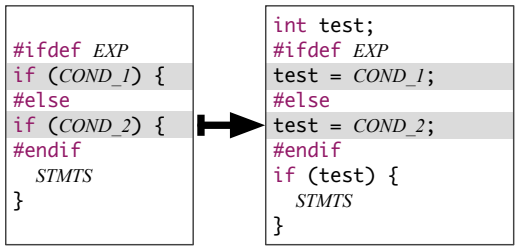
(→) `test` is not used in the code

3.3 Wrappers

Preprocessor directives are often used to wrap C statements in different ways. We address different forms of wrapping with Refactorings 3–5.

In Refactoring 3, we target another case of undisciplined preprocessor usage: alternative statements. We use an alternative `if` statement as an example, but there are similar refactorings for other alternative control-flow statements, such as `while` and `switch` statements. For this refactoring, we also need a fresh variable to keep the statement condition. Notice that this variable receives the evaluation of *COND_1* or *COND_2* depending on whether we define macro *EXP* is defined or not.

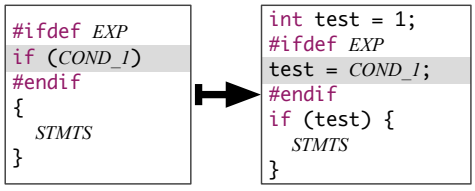
Refactoring 3: ⟨alternative `if` statements⟩



(→) `test` is not used in the code

In Refactoring 4, we present a refactoring to remove wrappers. Here, we also use a fresh variable to preserve the statement's condition and to discipline the preprocessor directive. We use an `if` wrapper as an example, but there are similar refactorings for removing undisciplined `while`, `for`, and `else-if` wrappers.

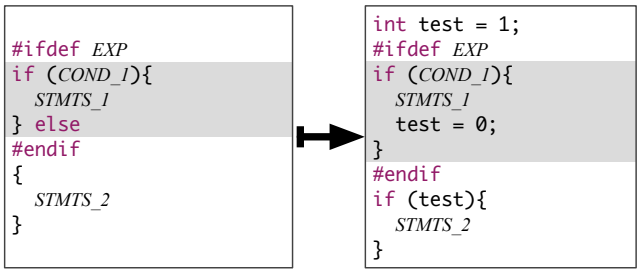
Refactoring 4: ⟨`if` wrapper⟩



(→) `test` is not used in the code

In Refactoring 5, we define a refactoring to remove `if` statements ending with an `else` statement. In this case, we replace the `else` by another `if` statement to resolve the undisciplined usage of the preprocessor. In this refactoring, the fresh variable `test` works like a flag to avoid executing *STMTS_2* when macro *EXP* is disabled.

Refactoring 5: ⟨`if` statements with an `else`⟩

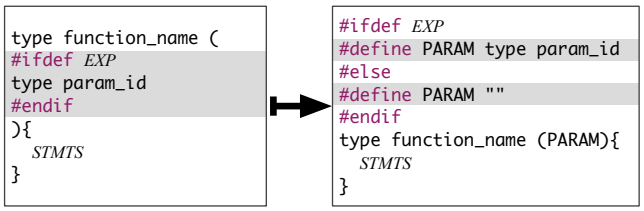


(→) `test` is not used in the code

3.4 Comma-Separated Elements

Refactoring 6 targets undisciplined preprocessor usage in comma-separated program elements. In this refactoring, we set a precondition that the original code does not define a macro `PARAM` or contains a token with that name, such as a type definition or identifier. If we change a macro definition that the original code is already using, we may introduce behavioral changes. This way, we modify the code locally without global impact. We handle other types of comma-separated elements, such as array and enum elements, with a similar refactoring.[3]

Refactoring 6: ⟨undisciplined function definitions⟩



(→) `PARAM` is not used in the code

4 Application Possibilities in Practice

In **RQ1** (number of possibilities), we investigate whether the undisciplined directives considered by our refactorings appear in real-world C projects, at all. To count the number of application possibilities for our refactorings in practice, we performed an analysis of 63 C popular projects from two sources: (1) 40 projects based on a corpus used in prior studies on the C preprocessor [24], [26], [10], [38], [31], [30], covering a range of different project sizes (2.6 thousand to 7.8 million lines of code); and (2) another 23 projects that use *GitHub* to submit patches to projects with the purpose of understanding the opinion of developers regarding our catalog (see **RQ2**). We selected our second corpus (23 projects) by

3. Our tool is able to handle undisciplined preprocessor directives with a list of comma-separated elements, not with only one as in Refactoring 6. The complexity with lists is that the tool needs to check where to put the comma that separates the elements, that is, one single comma between every pair of parameters.

searching for the most active projects on *GitHub*, considering the projects with the higher numbers of pull requests opened and closed. The reason was to get quick feedback from developers in terms of using undisciplined preprocessor directives to answer **RQ2** when submitting pull requests to the projects. Overall, our corpus includes projects from different domains, such as games, operating systems, text editors, and web servers, including *Bash*, *Gcc*, *Linux*, and *Vim*. Furthermore, our analysis considers both popular, big and mature projects, but also newer and smaller projects with less widespread use in practice.

To make the analysis scalable, we used *SrcML*[4] for identifying application possibilities for our refactorings. *SrcML* transforms C code into an XML representation, which we analyzed to detect the different patterns of undisciplined preprocessor directives. Table 1 presents the number of application possibilities for the 63 projects. Overall, we found 5670 application possibilities, showing that our refactorings are indeed relevant to real-world C projects (one refactoring possibility for every 3704 lines of code, on average).

According to our analysis, Refactorings 2 and 6 are the most frequently applicable in practice, while Refactorings 1 and 3 are the least frequent. Notably, we found that some projects heavily make use of undisciplined preprocessor directives, such as *Gcc*, *Glibc*, and *Vim*. But, there are also projects in which we have not found undisciplined directives, such as *Bison* and *Mpsolve*; others contain only a few undisciplined directives, such as *Libssh* and *Totem*. Overall, we found application possibilities in almost all projects analyzed (97%) in this study.

To better understand the characteristics of projects and the presence of undisciplined preprocessor directives, we analyzed the correlation between the number of application possibilities and the number of lines of code using Spearman's rank correlation, with 95% confidence interval. According to the results, Refactorings 2 and 6 show a moderate correlation. See Figure 3 with a scatter plot illustrating this correlation for Refactoring 6. For the other refactorings, Spearman's rank correlation test measures a weak correlation. For instance, in Figure 4, we show the correlation for Refactoring 2. We summarize all correlation results of the Spearman's rank correlation in Table 2. Furthermore, we found no correlation between the number of application possibilities and the number of developers using Spearman's rank correlation, and we have not found specific application domains with higher or lower numbers of undisciplined directives. Thus, it seems that the use of undisciplined preprocessor directives is more a preference of a few developers, who might try to avoid duplicating language tokens or who do not want to introduce local variables because of resource constraints, as we discuss in Section 6.1. To address undisciplined preprocessor directives in these situations, developers can modularize the source code into different functions or files and include them only in specific configurations of the source code, that is, developers can use different levels of granularity when dealing with variability [20], [30].

4. http://www.srcml.org/

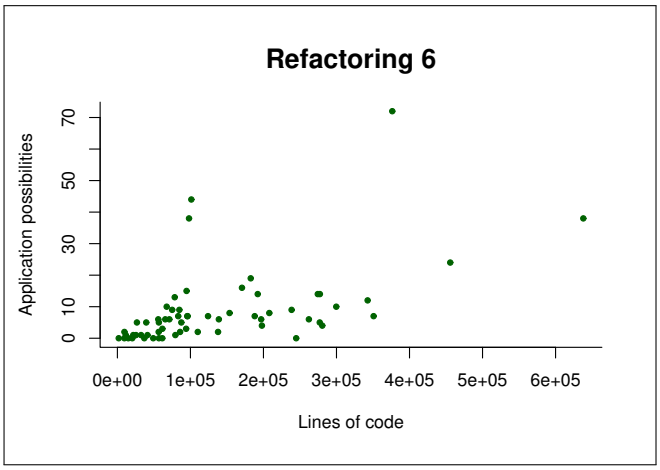


Fig. 3: Scatter plot showing the correlation between the number of application possibilities and the number of lines of code for Refactoring 6.

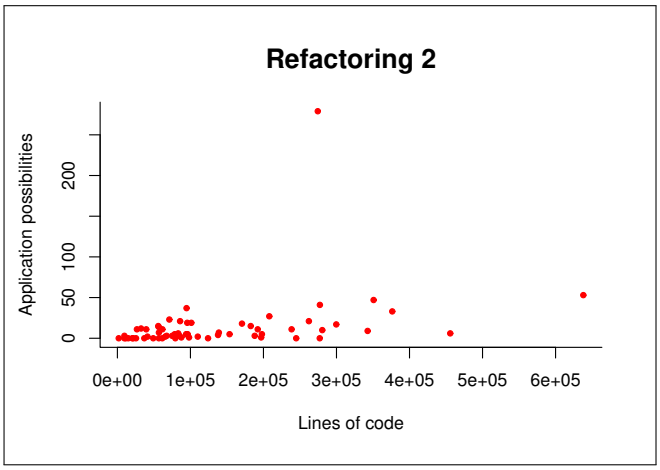


Fig. 4: Scatter plot showing the correlation between the number of application possibilities and the number of lines of code for Refactoring 2.

SUMMARY

We found application possibilities in almost all subject projects (97%), one application possibility for every 3704 lines of source code, on average, showing that developers indeed frequently use undisciplined directives that we can refactor in practice. Still, there are developers that prefer to use undisciplined preprocessor directives in specific situations, such as avoiding adding local variables because of resource constraints.

5 TOOL SUPPORT

We answered **RQ1** (number of possibilities) by using *SrcML* to detect undisciplined directives based on XML representations. Although, *SrcML* allowed us to scale

TABLE 1: Application possibilities in 63 C projects.

Project	Version	LOC	Domain	R1	R2	R3	R4	R5	R6
Angband	4.0.4	79 370	game	0	0	1	1	0	1
Amxmodx	1.8.3	262 186	server administration tool	0	21	7	12	84	6
Asfmapready	3.2.1	244 817	command line tools	0	0	3	0	0	0
Bash	4.2	96 153	command language interpreter	2	5	26	12	6	7
Berkeley DB	4.7.25	170 570	database system	5	18	6	1	9	16
Bison	2.0	20 379	parser generator	0	0	0	0	0	0
Busybox	1.23.1	182 555	common UNIX utilities	20	15	6	20	4	19
Cherokee	1.2.101	56 832	Web server	0	7	1	2	23	0
Clamav	0.97.6	342 636	antivirus software	9	9	4	4	17	12
Collectd	5.5.0	94 105	system administration tool	0	5	2	0	1	3
Curl	7.46.0	95 646	data transferring tool	5	19	2	8	38	7
Cvs	1.11.17	71 128	version control system	4	23	7	14	26	6
Dmd	2.069.2	94 687	language interpreter	2	37	12	9	2	15
Emacs	24.4	277 263	text editor	20	41	9	24	34	14
Ethersex	0.1.2	61 611	processor firmware	5	11	30	11	5	3
Freeradius	3.0.10	101 242	radius server	0	19	1	4	21	44
Gawk	3.1.4	39 499	interpreter	0	11	4	7	32	5
Gcc	4.9.2	2 927 120	compiler	51	371	32	172	121	114
Glibc	2.20	638 002	C library	29	53	16	76	71	38
Gnumeric	1.12.20	277 068	spreadsheet program	4	0	1	1	0	5
Gnuplot	4.6.1	83 260	plotting tool	2	6	4	15	42	7
Irssi	0.8.15	49 085	chat client	0	0	3	1	4	0
Kerberos	1.14	280 532	network authentication protocol	0	10	4	3	3	4
Kindb	1.0	61 486	database system	0	0	7	0	2	0
Hexchat	2.10.2	56 764	chat client	0	0	5	2	2	5
Libdsmcc	0.5	1814	DVB library	0	0	0	0	0	0
Libpng	1.5.14	32 432	PNG library	5	12	9	5	23	1
Libsoup	2.41.1	36 844	SOUP library	0	0	0	0	0	0
Libssh	0.5.3	25 451	SSH library	0	0	0	0	1	1
Libxml2	2.9.0	207 996	XML library	1	27	6	5	57	8
Linux	3.18.5	9 771 439	operating system kernel	129	60	40	71	277	518
M4	1.4.17	9623	macro expander	0	3	4	5	15	2
Machinekit	0.1	197 887	machine control platform	0	5	3	1	3	4
Mapserver	7.0.0	137 808	Web application framework	0	4	4	2	5	2
Mongo	1.1.8	41 185	MongoDB client library	0	2	0	0	1	1
Mpsolve	2.2	9562	mathematical software	0	0	0	0	0	0
Opensc	0.15.0	124 099	smart card tools and middleware	0	0	3	0	0	7
Openssl	1.0.2	238 529	SSL library	3	11	23	8	114	9
Opentx	2.1.6	188 001	radio transmitter firmware	0	3	1	1	2	7
Openvpn	2.3.6	56 478	virtual network tool	8	14	5	5	20	2
Ossec-hids	2.8.3	78 491	intrusion detection system	0	5	12	4	9	13
Pacemaker	1.1	87 540	cluster resource manager	0	1	0	0	0	5
Parrot	7.0.2	98 080	virtual machine	0	1	0	1	0	38
Pidgin	2.10.11	299 686	chat client	11	17	2	2	4	10
Prc-tools	2.3	14 945	gcc for Palm OS	1	0	0	0	0	0
Privoxy	3.0.19	26 730	proxy server	2	11	12	7	9	5
Python	2.7.9	376 373	language interpreter	34	33	12	14	49	72
Rcs	5.7	11 443	revision control system	2	0	0	0	0	1
Retroarch	1.2.2	192 152	libretro API	9	11	11	8	23	14
Sendmail	8.14.6	85 833	mail transfer agent	5	21	16	3	9	2
Sleuthkit	4.2.0	196 841	command line tools	0	1	5	0	15	6
Sqlite	3080200	138 896	database system	8	7	4	5	17	6
Syslog-ng	3.7	65 554	log management application	0	2	1	0	0	6
Sylpheed	3.3.0	110 047	e-mail client	13	2	3	0	2	2
Taulabs	20150922	455 769	autopilot system library	0	6	3	3	8	24
Tk	8.6.3	153 576	widget toolkit	2	5	2	1	0	8
Totem	2.17.5	21 716	video application	0	0	0	0	1	1
Uwsgi	1.9	84 843	application container	0	3	0	0	3	9
Vim	6.0	274 384	text editor	62	279	46	82	365	14
Wiredtiger	2.6.1	75 005	data management platform	0	3	0	0	0	9
Xfig	3.2.4	67 483	vector graphics editor	1	3	0	0	20	10
Xorg-server	1.9.3	350 817	window system	14	47	14	20	48	7
Xterm	224	55 966	terminal emulator	2	15	1	9	1	6
Total		**21 065 317**		**470**	**1295**	**435**	**661**	**1648**	**1161**

the analysis, it has some limitations that make the implementation of a reliable refactoring infrastructure infeasible, because *SrcML* uses heuristics and fails from time to time.

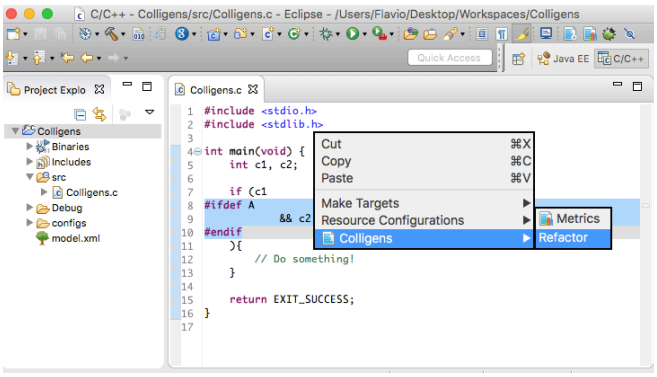


Fig. 5: Refactoring code with undisciplined preprocessor directives using *Colligens*.

To apply the refactorings automatically in a reliable fashion, we implemented our refactorings in *Colligens*,[5] a *plugin* for the *Eclipse* platform. *Colligens* resolves undisciplined preprocessor directives by applying the refactorings presented in the previous section. Figure 5 depicts a *Colligens* screen shot to refactor a code snippet with undisciplined conditions, as presented in Refactoring 2.

Colligens is built on top of *TypeChef* [23], a variability-aware parser that generates abstract syntax trees enhanced with variability information. *TypeChef* handles interactions of macros, file inclusion, and conditional compilation in a reliable fashion. Figure 6 (a) presents an abstract syntax tree generated from an `if` statement with an undisciplined preprocessor directive. The abstract syntax tree contains all variability information, that is, the `if` conditions generated when we enable and disabled preprocessor macro `A`. Notice that there is a choice node `A` that controls both configurations.

To detect application possibilities for our refactorings, we use the abstract syntax tree generated by *TypeChef* [23]. To detect an application possibility for Refactoring 2, for example, we search for `if` statements that contain optional nodes inside their condition. This way, we can transform the abstract syntax tree by applying Refactoring 2. Figure 6 (b) presents the resulting abstract syntax tree after applying Refactoring 2. We use a similar strategy for the other refactorings of our catalog.

5. *Colligens* is an open-source tool, distributed with *FeatureIDE* [47]. The tool is available for download at http://fosd.net/fide/.

TABLE 2: The correlation between the number of lines of code and the number of application possibilities using Spearman's rank correlation.

Refactoring	Confidence Interval	p-value	Correlation
R1	95%	0.00122	0.41
R2	95%	3.371e-05	0.50
R3	95%	0.000424	0.44
R4	95%	0.001121	0.41
R5	95%	0.001299	0.40
R6	95%	8.238e-09	0.66

6 Opinion of Developers

To answer question **RQ2** (opinion of developers) and to learn about the opinion of developers regarding our catalog of refactorings, we performed a survey among 246 developers (Section 6.1). Furthermore, we submitted 28 patches to 28 distinct real-world C projects to convert undisciplined directives into disciplined ones (Section 6.2). With our survey, we want to check the preferences of developers, verifying whether they keep undisciplined directives in the code or whether they apply our refactorings in practice. With the submission of patches, we want to check whether developers accept to change the source code only to remove undisciplined directives, that is, without introducing any bug fixes or new functionalities. We triangulate the results from the survey and patch submissions to get more confidence in our findings.

6.1 Survey

We performed an online survey among 246 developers. It is not the survey discussed in Section 2 and published in our previous study [30], which we performed to quantify the results of our interviews. To recruit participants for our new survey, we collected information about developers by mining the software repositories of several popular C projects, including the *Linux Kernel* and *Apache*. We randomly selected a number of developers from each project, and we sent 1832 emails asking developers to fill in our survey. Overall, 246 (7.9%) developers completed the survey. The majority of developers (75.2%) answered that they use the C preprocessor for at least 5 years, 15% have between 3–5 years of experience, and 9.8% use the preprocessor for less than 3 years. Furthermore, we found that 90% of the participants work both in closed-source and open-source projects, while 10% work only in open-source projects.

As the core of the survey, we asked developers six questions. We presented six pairs of two equivalent code snippets to our participants: (1) the original code from a real C project; and (2) the refactored version of the original code, created by applying one of our refactorings. We changed the order of the code snippets in the questions, that is, we put the refactored code at the right-hand and left-hand sides in different questions of our survey. For each pair of code snippets, we asked developers about their preferences. We used text boxes after each question so that developers could explain their choices. Specifically, we asked them about the following code fragments, which we list and discuss next. Notice that we used the same code fragments for all developers that answered our survey. It was just infeasible to customize the survey questions to every specific developer, due to the high number and diversity of developers and projects.

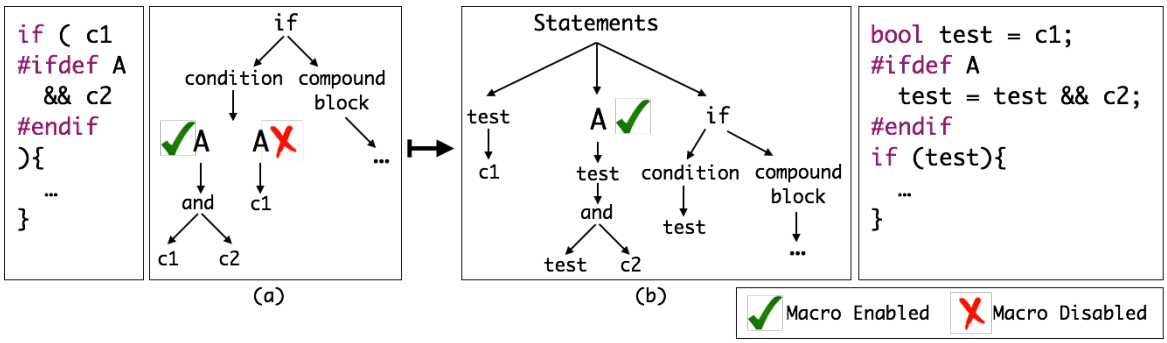


Fig. 6: (a) Abstract syntax tree with variability information. (b) Application of Refactoring 2.

We present a concrete instance of Refactoring 1 in Figure 7 (a), which shows an excerpt of *Vim's* source code with undisciplined preprocessor directives. In Figure 7 (b), we present the refactored (i.e., disciplined) version of the code. In our survey, 90.3% of developers preferred the refactored version of the code snippet. Developers mentioned that the refactored code does not break parts of statements, even duplicating a few language tokens, and it is the only one that works for function-like macros. This result aligns with the results of our patch submission that we discuss in Section 6.2, in which developers of *Libpng* asked us to duplicate a few language tokens before accepting our patch.

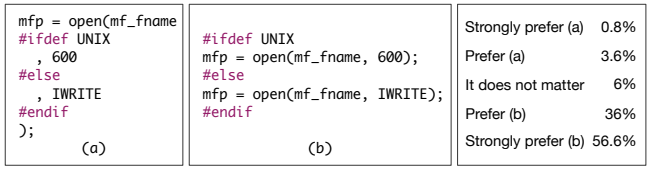


Fig. 7: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

In Figure 8, we present a pair of code snippets with an application of Refactoring 2. Figure 8 (a) shows part of *Libpng's* source code with undisciplined preprocessor usage. Figure 8 (b) presents the refactored version of the code. In our online survey, 70.4% of developers preferred the refactored version. Regarding the 4.4% of developers that prefer the original code presented in Figure 8 (a), 13% stated that they wanted to avoid extra local variables, which is important for devices with memory constraints. This result aligns with the results of our patch submission that we discuss in Section 6.2, in which developers of *Ethersex* also complained about limited resources and extra local variables.

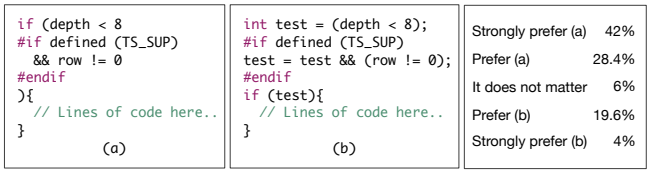


Fig. 8: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

We presented an application of Refactoring 3 to developers using a code snippet of *OpenSSL*, as illustrated in Figure 9 (a). In Figure 9 (b), we show the refactored code. Based on the answers of 246 developers, almost 65% preferred the refactored version. Considering the 26% of developers that prefer the original code presented in Figure 9 (a), 11% of them argued that the refactored code requires more memory. Furthermore, when considering the 9.2% of developers that answered that it does not matter, 4% of them stated that it does not matter because both versions are readable.

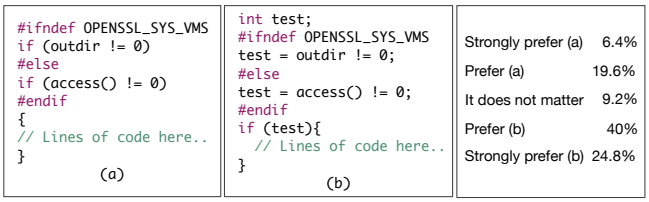


Fig. 9: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

In Figure 10 (a), we show a code snippet of *Gcc* with an application possibility of Refactoring 4. In Figure 10 (b), we present the refactored code. According to the answers of developers, 62% preferred the refactored version. Some developers (4%) stated that they have no preference, saying that there is not much difference in this small code snippet. Moreover, a few developers (3%) mentioned that they prefer the original version because of resource constraints due to the use of local variables. Some compilers will optimize the source code and it will not be a problem. However, developers argued that they cannot rely on that because some compilers might not optimize the source code [30].

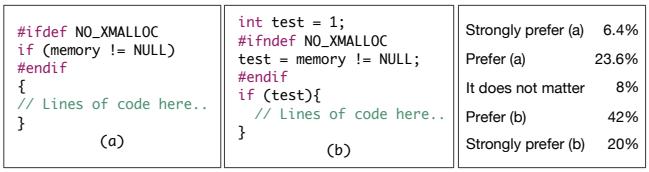


Fig. 10: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

In Figure 11 (a), we show a code snippet of *BusyBox* with an application possibility of Refactoring 5. In Figure 11 (b), we show the refactored code. According to the answers of developers, almost 54% preferred the refactored version. Regarding almost 36% of developers that prefer the undisciplined version of the code, 17% stated that the logic of the refactored code is not simple.

```
#ifdef VT_OPENQRY
if (ioctl(STDIN) != 0){
  // Lines of code here..
} else
#endif
if (!s)
   putenv((char*) TRML);
```

(a)

```
int test = 1;
#ifndef VT_OPENQRY
if (ioctl(STDIN) != 0){
  // Lines of code here..
  test = 0;
}
#endif
if (test){
  if (!s)
     putenv((char*) TRML);
}
```

(b)

Strongly prefer (a)	9.6%
Prefer (a)	26%
It does not matter	10.8%
Prefer (b)	38.8%
Strongly prefer (b)	14.8%

Fig. 11: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

Last, we show an instance of Refactoring 6 in Figure 12, which was taken from *Vim*. Figure 12 (b) presents the refactored version of the code snippet. From the developers that completed our survey, 44.8% preferred the refactored version. However, 38.8% of the developers preferred the undisciplined version, most of them (12%) stating that the additional macro might leave the source code more difficult to read.

```
void msgW32(
#if defined (GUI_W32)
  Xt client,
#endif
Id *id){
  // lines of code..
}
```

(a)

```
#if defined (GUI_W32)
#define PARAM Xt client,
#else
#define PARAM
#endif
void msgW32(PARAM Id *id){
  // Lines of code..
}
```

(b)

Strongly prefer (a)	8%
Prefer (a)	30.8%
It does not matter	16.4%
Prefer (b)	29.2%
Strongly prefer (b)	15.6%

Fig. 12: (a) Original code with undisciplined preprocessor directives. (b) Refactored code.

Overall, based on the answers of developers, we conclude that developers agree with most strategies of resolving undisciplined directives. The repetition of a few tokens, as we illustrate in Figure 7, received support from more than 90% of the surveyed developers, while the use of preprocessor macros, as presented in Figure 12, received the weakest support. So, developers prefer to resolve undisciplined directives by duplicating a few tokens and by adding a local variable, in cases where resource constraints are not a problem. The results of our patch submission that we present in Section 6.2 complement and support the results of our survey. However, developers avoid using macros as they might leave the source code more difficult to read and understand, especially when defined and used in different files.

SUMMARY

In six scenarios from our subject projects, most (63%) of the 246 surveyed developers preferred to use the refactored (i.e., disciplined) code, instead of using the preprocessor in undisciplined ways. More than 90% of the developers support the repetition of a few tokens to make preprocessor directives disciplined, but they prefer to avoid using additional macros.

6.2 Submitting Patches

To further understand the real-world relevance of our refactorings (RQ2), we submitted 28 patches to different popular C projects. We want to know whether undisciplined directives are important enough to motivate developers to change code only to remove these directives. For the selection of subject projects, we used *GHTorrent* [17], with the goal of identifying active projects that heavily use pull requests on *GitHub*. We submitted patches to the 28 most active projects with application possibilities for our refactorings. Overall, developers accepted 21 (75%) patches, one each in: *Angband*; *Amxmodx*; *Asfmapready*; *Collectd*; *Curl*; *Dmd*; *Libpng*; *Linux*; *Mapserver*; *Machinekit*; *Mongo*; *Opensc*; *Openssl*; *Opentx*; *Ossec-hids*; *Retroarch*; *Sleuthkit*; *Syslog-ng*; *Taulabs*; *Uwsgi*; and *Wiredtiger*.

We submitted the patches via *GitHub*, which allows developers to include comments on patches. Besides the commit messages, developers can interact by sending messages to each other before accepting or rejecting a patch submission.[6] We submitted one refactoring converting an undisciplined into a disciplined directive per patch, and we submitted one patch per project. Thus, our patches were judged by a broad audience of developers. This way, we also avoided the problem of having many patches evaluated by the same few developers.

The feedback we received supports the perception that undisciplined preprocessor usage influences the code quality negatively. For most patches, developers agreed with our suggestion to resolve undisciplined preprocessor usage. For example, one developer mentioned that the refactoring *"makes sense [to him] and it is a good idea."* Developers accepted 12 (33%) patches without asking for changes. However, for some patches, developers asked, for example, to rename local variables or to include or exclude spaces between brackets, to better follow the project's standards. For instance, one developer said that *"[the patch] would be fine except for the unnecessary extra parentheses."* Table 3 presents the patches developers accepted after we applied a few minor changes.

While many patches have been accepted directly or after minor modifications, we also noticed some resistance or different expectations for refactorings in a few projects. Developers argued that our patches made the

6. For a discussion about undisciplined directives with developers of the *Nipy* project, see https://github.com/nipy/nipy/pull/384.

code less readable and asked us to move the preprocessor directives to separate helper functions and to extract the directives to a macro function. In Figure 13, we show the strategy we used to refactor an `if` statement with an undisciplined condition in the *Syslog-ng* project, instead of applying Refactoring 2. Notice that a helper function works well for this specific situation, but this refactoring is not suitable for all undisciplined conditions, that is, it requires many extra functions.

TABLE 3: Patches accepted after minor changes.

Project	Changes requested by developers
Dmd	Remove unnecessary parentheses
Linux	Fix typo
Libpng	Duplicate the code instead of adding a new local variable
Machinekit	Fix indentation
Openssl	Rename local variable
Opentx	Remove unnecessary parentheses
Retroarch	Use integer instead of boolean
Syslog-ng	Extract code to a helper function
Uwsgi	Extract directives to a macro

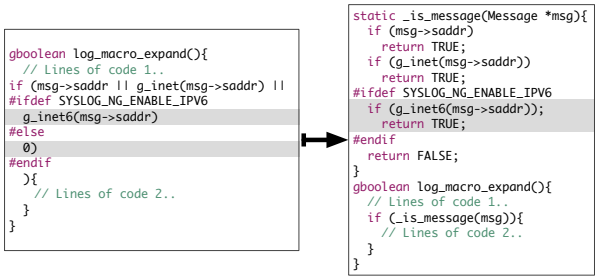


Fig. 13: Feedback we received from developers when submitting a patch to *Syslog-ng*.

We also noticed that some developers are resistant to apply any changes to their source code. Such developers raised some reasons, saying that "we know that [the code] works, and a change there would need very close scrutiny to ensure [that] no combination of features gets broken, review time needed." In another project, developers complained about introducing local variables. For example, a developer said that "I agree with you [that we should avoid undisciplined directives], but the resources are limited [in our context] and we should make every effort to not waste them." So, they did not accept our patch because of the new local variable that we use to discipline the preprocessor directives. Table 4 lists the 7 patches rejected by developers.

TABLE 4: Patches rejected.

Project	Argument against changes
Ethersex	The patch defines a new local variable; we have limited resources
Freeradius	The patch needs improvements; it is harder to read
Hexchat	The new code is harder to read
Kerberos	The code is old; what we need is to remove the conditional directives
Irssi	The patch needs to be improved
Openvpn	The code is working and changes will require test effort and time
Pacemaker	The changes require test effort and time

In summary, we submitted patches covering all six types of refactorings: we submitted 6 patches using Refactoring 1 and all patches were accepted; 5 patches submitted using Refactoring 2 and developers accepted 80% of the patches; 5 patches submitted using Refactoring 3 and 4 patches (80%) accepted; 5 patches submitted using Refactoring 4 and developers accepted 60% of them; 3 patches submitted using Refactoring 5 and one (33%) accepted; and 4 patches submitted using Refactoring 6 and 2 (50%) accepted.

SUMMARY

We received positive feedback from developers when submitting patches to resolve undisciplined directives in 28 projects. Overall, developers accepted 21 (75%) out of the 28 patches submitted, demostrating that developers think that undisciplined directives are important enough to make changes in the source code only to remove these directives.

7 Behavior Preservation

The C preprocessor hinders the development of tool support that is typically available for other languages, such as automated refactoring [49], [23], [15], [29], [21], [48]. After applying refactorings in C, developers need time to manually review the different configurations of the source code. Thus, it is important to make sure that the refactorings of our catalog do not introduce behavioral changes.

In previous work [32], we already verified that the Refactorings 1 and 6 are behavior preserving. These refactorings do not introduce local variables and allow us to prove behavior preservation purely syntactically. In a nutshell, we preprocessed and compared the original and refactored codes syntactically, that is, showing that the result of preprocessing the original and refactored code is always equivalent for all configurations. However, the other refactorings introduce local variables, requiring a more sophisticated strategy to check behavior preservation.

To answer question **RQ3** (behavior preservation) and to gain confidence into our refactorings, we started by analyzing a subset of application possibilities by using manual code reviews. Furthermore, we used automated testing. Previous studies have found many bugs in refactoring engines [42], and the difficulties to deal with several configurations make refactorings in configurable systems more error prone. We performed a multi-method evaluation that considers two perspectives to triangulate the results and to get more confidence in our findings: (1) programs automatically generated based on a model of a subset of the C language, which we developed using *Alloy*;[7] and (2) real-world projects with test cases available. We found some problems related to

7. http://alloy.mit.edu

compiler-specific issues in previous studies [50]. Thus, we decided to run our analysis on different operating systems, that is, *Linux* and *Mac OS*.

7.1 Refactoring Generated Programs

To test behavior preservation, we use differential testing by running test cases before and after applying the refactorings. We have not used a formal approach to verify behavior preservation, which is definitely worth. There are some formal approaches to check behavior preservation in refactorings [6], [39]. Some approaches formalize refactorings for a subset of the language [6], while others consider the complete language [3], but contains bugs [41]. However, notice that we cannot provide a final prove for `RQ3`, as we did not perform a formal approach. Thus, we provide empirical evidence that improves confidence that our refactorings are behavior preserving.

The specification and proof of refactorings is a time-consuming and non-trivial task, in particular because of the semantic complexities of the languages, such as C and Java. Our approach, which uses differential testing, is less time-consuming and it has been used successfully to detect bugs in refactorings of traditional tools, such as *Eclipse* and others, which defined refactorings based on formal specifications [41]. The approach used by Soares et al. [41] is similar to ours, but they focused on single systems, that is, programs without configuration options. In our case, we focus on program families, i.e., programs with many configuration options. For instance, in a program with one configuration option, it can be enabled or disabled. In this sense, we need to check behavior preservation in both configurations: (1) with the macro enabled; and (2) with the macro disabled. The number of configurations grows exponentially when increasing the number of configuration options. Proving a refactoring sound for a single program is considered a challenge [40], for program families it is even more difficult as we need to handle several configurations. We discuss the steps we performed to check behavior preservation next, as shown in Figure 14.

In *Step 1*, we create a model of a subset of the C language in *Alloy* to generate configurable programs (i.e., *A*, *B*, and *C*) with an opportunity to apply our refactorings. Appendix A presents more information about the C model. We specified only a subset of the C language formally because there are parts of the language that we do not use directly in our refactorings, such as structures, pointers, arrays, Strings, Enumerators, input and output, and operations with files. So, we ignored those aspects, which would increase the complexity of the model. In *Step 2*, we select each program generated previously (e.g., program *A*) and use the preprocessor to create all different configurations of that specific program. In Figure 14, we show the two possible configurations of program *A*: (*C1*) with macro `EXP` enabled; and (*C2*) with macro `EXP` disabled. Then, for each configuration of the generated programs, in *Step 3*, we generate unit test cases automatically by using a test case generator for C programs [34]. In *Step 4*, we apply a refactoring of our catalog to each program generated previously using *Colligens*. For example, considering our example program *A*, it generates an equivalent program *A'* without undisciplined preprocessor usage. In *Step 5*, we use the preprocessor to generate each possible configuration for the refactored programs (i.e., *C1'* and *C2'*). In *Step 6*, we run the test cases on the original and refactored programs to search for behavioral changes. For instance, the output of a test case for program *A*, with macro `EXP` enabled, must be the same as the output for program *A'*, with macro `EXP` enabled, giving the same input value for both programs. The same must hold for all configurations of the generated programs.

In Figure 15 (a), we list a program generated with the possibility to apply Refactoring 2. The preprocessor directives at Lines 11 and 13 split up parts of the `if` condition. In Figure 15 (b), we present the code snippet generated after applying Refactoring 2. Notice that our strategy generates small programs like the one presented in Figure 15 (a), but we considered different C operators, types, and initial values. This way, we can use a *brute-force* approach to test for behavioral changes.

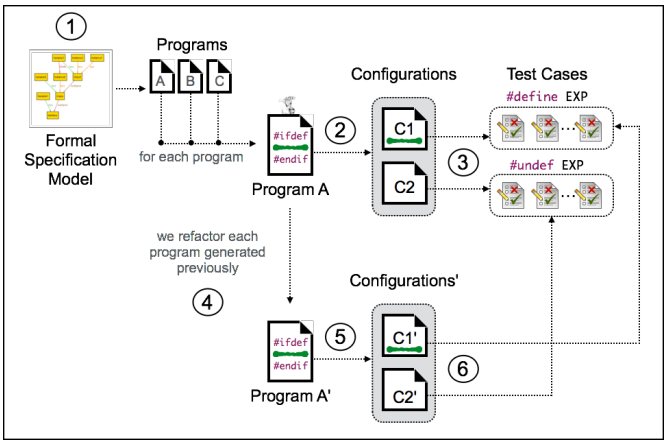


Fig. 14: Applying differential testing to verify behavior preservation of refactorings.

Table 5 presents the results obtained from generating 10K programs for each refactoring. Regarding Refactoring 4, which contains two variations, we generated 10K programs for each. According to the results, our model generated up to 84% of valid programs (according to the C standards). The reason that the model generates invalid programs, which do not compile, is that we have not included all the necessary predicates and clauses to avoid Alloy generating invalid programs. The effort to do that is pretty high, and it will not change the results of our behavior preservation analysis because we considered only the valid program that compile successfully using *gcc*. We applied our refactorings to all valid programs, and we introduced no compilation errors after applying the refactorings. Overall, we detected

13 behavioral changes: five behavioral changes caused by a conceptual problem in Refactoring 2, and eight behavioral changes caused by bugs in the implementation of our refactorings.

To fix the behavioral changes introduced by our refactoring, we defined a new version of Refactoring 2, as already presented in Section 3. In Figure 16, we present the initial refactoring that introduces behavioral changes. The problem with this initial refactoring was that the C language does not specify the order of precedence when evaluating expressions with Boolean operators. Because of this unspecified behavior, different compilers may evaluate `if` conditions differently. We detected this problem when verifying the behavior of the program presented in Figure 15 (a). When running this program on *Linux* with *Gcc*, the compiler evaluates the function call (`F1`) at Line 12 before evaluating variable `Global0` at Line 10. On the other hand, *Gcc* evaluates variable `Global0` first when running the program on *Mac OS*. Notice that, by applying the initial refactoring shown in Figure 16, variable `Global0` is always evaluated before calling function `F1`, as we can see in Figure 15 (b). This way, the initial refactoring introduces a behavioral change when running the program on *Linux*.

```
1. int Global0 = 1;
2.
3. float F1(float P0){
4.   Global0 = 0;
5.   return P0;
6. }
7.
8. float F0(float P0){
9.   float Local0 = 1;
10.  if (Global0
11.#ifdef TAG
12.       & F1(P0)
13.#endif
14.  ){
15.    Local0 += 9;
16.    return P0;
17.  }
18.  return P0;
19.}
```

(a)

```
1. int Global0 = 1;
2.
3. float F1(float P0){
4.   Global0 = 0;
5.   return P0;
6. }
7.
8. float F0(float P0){
9.   float Local0 = 1;
10.  bool test = Global0;
11.#ifdef TAG
12.  test = test & F1(P0);
13.#endif
14.  if (test){
15.    Local0 += 9;
16.    return P0;
17.  }
18.  return P0;
19.}
```

(b)

Fig. 15: **(a)** Example of generated program. **(b)** Example of refactored program.

Regarding behavioral changes caused by bugs in the implementation of our refactorings, we found five bugs in the pretty printer, which missed relevant white spaces between identifiers and operators, and three bugs related to the use of integer variables instead of Boolean variables in `if` conditions. Our catalog of refactorings uses the Boolean type as defined in the `stdbool` library. By using integer variables, the implementation of the refactorings caused behavioral changes when converting float values to integer. We fixed all these bugs in the current implementation. Furthermore, based on the feedback we received from developers during our survey, we updated Refactoring 5, as already presented in Section 3.

TABLE 5: Results of behavioral changes regarding the generated programs.

	R2	R3	R4	R5
Generated programs	8461	8557	17,491	7589
Behavioral changes	5	1	5	2
Behavioral changes (after fixes)	0	0	0	0

SUMMARY

By generating programs automatically using a model of a subset of the C language, we found and fixed a few behavioral changes introduced by our refactorings and a number of problems in the implementation of our catalog, 62% related to unspecified behavior in the C language. This way, we increase confidence that the refactoring implementation and descriptions are behavior preserving.

7.2 Refactoring Real-World Projects

In Section 7.1, we checked behavioral changes considering small programs generated automatically, which might not use the C preprocessor in the way that developers use it in real-world projects. Now, we want to apply the refactorings to code written by real developers, considering well-known and widely used projects.

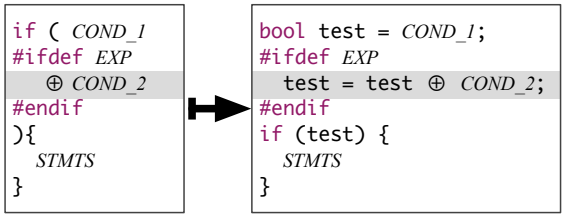


Fig. 16: Removing undisciplined `if` conditions.

To evaluate behavior preservation (RQ3) in real-world projects, we extended our refactoring implementation to use the *Morpheus* [25] refactoring testing infrastructure, which is also based on *TypeChef*. We use the *Morpheus* testing infrastructure to detect the configurations impacted by the refactorings instead of using the *brute-force* approach that we used in Section 7.1. As our case studies, we selected *BusyBox*,[8] *OpenSSL*,[9] and *SQLite*,[10] three projects already used with *Morpheus*, and with test cases available. *BusyBox* is a project that combines small versions of many common *UNIX* utilities into a single small executable. It contains 522 files and 19K lines of C code (version 1.18.5). *BusyBox* provides 792 configuration options implemented with preprocessor directives. *OpenSSL* implements secure Internet protocols, contains 733 files and 233K lines of C code. *OpenSSL* provides 589 configuration options. *SQLite* is a library implementing

8. http://www.busybox.net/
9. https://www.openssl.org/
10. https://www.sqlite.org/

a relational database management system, its code base consists only of two source-code files (amalgamation version 3.8.1), with 143K lines of C code, which can be configured using 93 configuration options.

We applied our refactorings to all 45 cases of undisciplined preprocessor usage in *BusyBox*, all 146 cases in *OpenSSL*, and all 33 cases in *SQLite*, as presented in Table 6. *BusyBox* comes with a test suite with 410 test cases for 74 files, out of which 46 tests fail on the original code (which we ignored during our evaluation). *OpenSSL* provides a test suite for each individual component, including the implementation of hashing functions (such as *MD5* and *SHA-256*) and key-generation and encryption algorithms. The test suite of *OpenSSL* does not indicate the exact number of test cases, but it provides an output message informing the failure or success of the complete test suite. For *SQLite*, we used the proprietary *TH3* test suite.

TABLE 6: Results of testing on *BusyBox*, *OpenSSL*, and *SQLite*.

	R2	R3	R4	R5
BusyBox	15	6	20	4
OpenSSL	11	23	8	114
SQLite	7	4	5	17
Behavioral changes	0	0	0	0

To test that our refactorings are behavior preserving, we applied the approach used by Liebig et al. [25]: We used two oracles: (1) the source code of our subject systems still compiles, and (2) the results of the test cases of the projects (pre-refactoring and post-refactoring) do not vary. To incorporate variability, we detected the configurations affected by the refactorings and test them. This functionality is performed by *Morpheus*, which also considers nested preprocessor directives. Notice that the *brute-force* approach that we used in Section 7.1 does not scale to real-world projects. So, we consider only the configurations affected by the refactorings. For example, assuming that a refactoring impacts options `A` and `B` in a source file with 5 configuration options. Thus, we select to test four configurations to make sure that all combinations of `A` and `B` are tested: (1) `A` and `B` enabled; (2) `A` and `B` disabled; (3) `A` enabled and `B` disabled; and (4) `A` disabled and `B` enabled.

After running the test cases before and after applying our refactorings in the three systems, we found no behavioral changes or implementation problems in our catalog of refactorings. We used print statements to check whether that the code impacted by the refactorings were covered by the test cases. We obtained the following percentages for the three projects: 58% for *BusyBox*; 47% for *SQLite*; and 63% for *OpenSSL*. Thus, it means that 42% of the refactorings that we applied in *BusyBox* were not covered by the test suite. In *SQLite*, 33% of the refactorings we applied were not covered by the test cases, and 37% of the refactorings in *OpenSSL* were not covered by the test cases also.

SUMMARY

By applying our refactorings in three real-world C projects (BusyBox, OpenSSL, and SQLite), we found no behavioral changes after removing 224 undisciplined directives in the three projects, further increasing confidence that the refactorings of our catalog and their implementation are behavior-preserving.

8 THREATS TO VALIDITY

Next, we discuss potential threats to validity of our studies, considering the opinion of developers, frequencies of application possibilities in practice, and behavior preservation. In our survey, we asked developers about their preferences using two equivalent code snippets. This way, we can only conclude that developers accept our strategies to resolve undisciplined directives. Furthermore, we did not use customized code snippets for every specific developer as we wanted to apply the survey to broad audience of developers. To minimize this threat, we used simple code snippets to make sure that all developers understand the code fragments.

We defined our catalog based on patterns of undisciplined preprocessor directives detected in 12 C open source projects [32], including *Apache*, *Gzip*, and *Lighttpd*. By using these refactorings, we removed all undisciplined directives of these projects. However, the catalog is not complete, and variations of our refactorings are necessary to remove all undisciplined preprocessor directives.

Regarding application possibilities in practice, we used an XML-based tool to detect application possibilities. *SrcML*[11] uses heuristics that may fail in source code with undisciplined preprocessor directives. To minimize this threat, we also determined the application possibilities of three projects (*BusyBox*, *Libssh*, and *Libpng*) using *TypeChef* [23], which works reliably in the presence of undisciplined preprocessor directives. *TypeChef* requires a time-consuming setup, though, hindering the analysis of all 63 projects. The numbers of application possibilities vary by two percentage points when comparing the results of *TypeChef* and *SrcML*.

We used a model of a subset of the C language to generate programs with refactoring possibilities automatically. Our model considers only a subset of the C language, though. Thus, we might miss behavioral changes caused by other C constructs that we have not considered. Furthermore, the undisciplined directives that we generate automatically might be different from the ones used in practice. To minimize this threat, we also used three real-world projects, *BusyBox*, *OpenSSL*, and *SQLite*, to test for behavior preservation. However, we found that some refactorings that we performed are not covered by the test suite of these projects, even using all test cases available: 42% of the refactorings we

11. http://www.srcml.org/

performed in *BusyBox*, 53% in *OpenSSL*, and 37% of the refactorings performed in *SQLite*.

9 Related Work

Opdyke defines a refactoring as a behavior-preserving program transformation [35]. To test for behavior preservation, Opdyke uses successive compilation and tests; his work focuses on refactorings of a single program. Most commercial refactoring tools today also focus on refactorings of single programs, including *Eclipse*, *Netbeans*, and *XCode* [25]. These tools provide no support to refactor C code with variability introduced by preprocessor directives.

The refactoring of C code is different from refactorings in other languages, due to the presence of the C preprocessor. In this context, we have a number of configuration options to tailor the program to different hardware platforms and application scenarios. So, ideally, a C refactoring tool has to consider all different configurations, which is a challenging task as the preprocessor allows developers to annotate arbitrary code fragments, such as an opening bracket without its corresponding closing one [12], [26], [25].

There are some approaches to refactor C code with preprocessor directives by using heuristics and limiting developers to annotate only disciplined annotations: conditional directives that surround only entire functions, type definitions, and statements [26]. For instance, Baxter and Mehlich proposed *DMS*, a source-code transformation tool for C/C++ [4]. In a more recent work, they emphasized the problems of using the preprocessor in undisciplined ways [5]. The *DMS* tool focuses on reverse engineering to gather design information and to ease maintenance tasks. Platoff et al. [37] also used the strategy of limiting developers to wrap only entire code blocks when using the *PTT* refactoring tool.

Other approaches use a variant-based strategy that preprocesses the code to generate different variants and apply a refactoring to each variant individually. Garrido and Johnson [13] developed *CRefactory*, a refactoring tool for C that considers different configurations in a variant-based fashion. *CRefactory* focuses on C refactorings, such as renaming functions and extracting macros [15]. Vittek et al. also use this strategy in *Xrefactory*, a refactoring browser for C, and discusses certain complications introduced by the preprocessor [49]. Spinellis et al. [45] also use a variant-based approach in *CScout*. They developed a Web-based, interactive front end to support the precise realization of rename and remove refactorings on the original C source code. Waddington and Yao proposed *Proteus* also using a variant-based approach. As variant-based refactorings consider only a single variant of the code at a time and refactor each variant individually, they introduce the overhead of merging all variants after applying the refactorings [25].

In recent work, Liebig et al. [25] proposed a variability-aware refactoring approach, which preserves the behavior of all variants of a configurable system. Liebig et al. uses variability-aware analysis, which considers all possible configurations of the source code at the same time [46]. Their study keeps all variability information, different from strategies that preprocess or modify the source code before parsing it [36], [43]. Liebig at el. demonstrated the applicability and scalability of their approach by implementing a sound refactoring engine (*Morpheus*) and by performing refactorings (Extract Function, Rename, and Inline Function) in three real-world projects: *BusyBox*, *OpenSSL*, and *SQLite*. They also provided evidence for the correctness of the refactorings implemented by running the original test cases of the projects before and after applying the refactorings.

In our study, we extended *Morpheus* by implementing our catalog of refactorings on top and by applying our refactorings in *BusyBox*, *OpenSSL*, and *SQLite*. Our work also uses a variability-aware approach, but it has a different focus. The related work discussed so far focus on refactorings such as rename, extracting functions, and extracting macros. We apply C refactorings to the preprocessor directives themselves, focusing on resolving undisciplined preprocessor usage. After resolving undisciplined preprocessor usage with our refactorings, we allow developers to use several C supporting tools that work only in the presence of disciplined annotations. Garrido and Johnson also proposed refactorings to convert undisciplined directives into disciplined ones [14], but the strategy used is naive and clone complete blocks of source code. In our study, we proposed specific refactorings for different types of undisciplined directives to minimize code cloning. Thus, we duplicate only a few language tokens, as we can see in Refactoring 1, for example, in which `COND_1` appears twice in the refactored code.

To implement variability-aware refactoring engines, we and Liebig et al. [25] used a variability-aware parser (i.e., *TypeChef*, by Kästner et al. [23]). *TypeChef* analyzes all possible configurations of a piece of C code, and also performs type checking [22] and data-flow [27] analyses. Gazzillo and Grimm [16] proposed another variability-aware parser (*SuperC*).

Other studies investigate refactorings of preprocessor directives into aspects. Adams et al. [1] analyzed the feasibility of refactoring `#ifdefs` to aspects; they did not implement any tool to perform the refactorings automatically. According to their work, it is possible to refactor the majority of preprocessor directives into aspects. Lohmann et al. [28] refactored the *eCos* operating system kernel using AspectC++, an Aspect-Oriented Programming (AOP) extension to the C++ language, and analyzed the runtime and memory costs of aspects. In another study, Batory et al. [3] proposed a novel implementation mechanism [2] based on feature modules, which allows developers to create programs by adding features. Our work also focuses on refactorings of preprocessor directives, but we refactor the directives without introducing another variability implementation mechanism such as aspects and feature modules.

Borba et al. [7] defined a theory to refactor software families. They applied specific artifacts, such as feature models and configuration knowledge, and proposed a theory to test behavior preservation. Furthermore, Borba et al. developed a theory using a formal specification language and proves some compositionality properties of this theory.

Ferreira et al. [11] present an implementation of Borba's software family theory. It proposed tools to evaluate if a software family transformation preserves behavior. These tools use test cases to minimize the chances of introducing behavioral changes with refactoring. They are based on *SafeRefactor*, which creates and runs test cases automatically to increase confidence that a transformation preserves behavior [42]. In our work, we check our refactorings to avoid behavioral changes by using a similar strategy as in *SafeRefactor*, proposed by Mongiovi et al. [34], which extended *SafeRefactor* to the C language.

10 Concluding Remarks

We evaluated a catalog of refactorings to convert undisciplined directives into disciplined ones. In particular, we evaluated the refactorings regarding the opinion of developers, number of application possibilities in practice, and behavior preservation.

Our results reveal that most developers prefer to use the disciplined version of the code instead of the original code. By analyzing 63 real-world and popular C projects, including *Gcc*, the *Linux Kernel*, and *Python*, we found 5670 opportunities to apply our catalog of refactorings. We have shown that undisciplined preprocessor directives appear in many well-known and widely used C projects. Even projects with explicit coding guidelines targeting undisciplined directives, such as the *Linux Kernel*, developers introduced many undisciplined directives in practice. This finding suggests that code guidelines are not enough, and developers need new tools to check coding guidelines strictly and to reject patches that do not follow the guidelines of the projects. We submitted 28 patches to convert undisciplined directives into disciplined ones and developers accepted 21 (75%) patches.

We found that developers support the idea of converting undisciplined directives into disciplined ones with our patch submissions. However, it is also clear that some developers prefer to refactor the source code when making other necessary changes in the code, for example, to fix bugs. Especially, as developers are aware of the difficulties of testing the different configurations of the source code [30]. So, our results show that we need better integration between refactoring tools and software repositories. As a consequence, refactoring tools should suggest changes directly to developers that just changed code that needs improvements regarding undisciplined preprocessor usage, as soon as developers submit a new commit.

To verify behavior preservation, we applied our refactorings in more than 36 thousand programs generated automatically using a formal model, but also in three real-world projects: *BusyBox*, *OpenSSL*, and *SQLite*. We detected and fixed a few behavioral changes introduced by our refactorings, 62% caused by unspecified behavior in the C language, and a number of problems in the implementation of our catalog of refactorings. Previous studies have found bugs in many refactoring engines [42], such as *Eclipse* and *JRRT*, showing the complexity of refactoring code automatically. So, our results support that we need to improve the state-of-the-art regarding refactoring engines. In particular, refactorings in C are more challenging as developers use preprocessor directives, creating different configurations of the source code, which needs to be checked separately regarding behavior preservation.

In future work, we plan to perform more studies to better understand the effects of using undisciplined preprocessor directives, including different types of studies, such as controlled experiments, surveys and interviews. In particular, there is an opportunity to perform a new survey, but now considering customized code snippets for every specific developer. This way, we can use more complex code snippets that developers are familiar with to get more evidence about the problems of using undisciplined preprocessor directives. Furthermore, we also intend to extend our C model to generate more complex programs to verify behavior preservation in our refactorings to improve confidence in our findings.

Acknowledgement

This work has been partially supported by CNPq 460883/2014-3, 465614/2014-0, and 306610/2013-2, FAPEAL PPGs 14/2016, CAPES 175956 and 117875, and DEVASSES, funded by the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no PIRSES-GA-2013-612569. Kaestner's work has been supported by the NSF awards 1318808 and 1552944, the Science of Security Lablet (H9823014C0140), and AFRL and DARPA (FA8750-16-2-0042). Apel's work has been supported by the German Research Foundation (AP 206/4, AP 206/5, and AP 206/6).

References

[1] Adams, B., De Meuter, W., Tromp, H., Hassan, A.E.: Can we refactor conditional compilation into aspects? In: Proceedings of the ACM International Conference on Aspect-Oriented Software Development. pp. 243–254. ACM (2009)

[2] Apel, S., Batory, D., Kstner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer (2013)

[3] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the International Conference on Software Engineering. pp. 187–197. IEEE (2003)

[4] Baxter, I.: Design maintenance systems. Communication of the ACM 35(4), 73–89 (1992)

[5] Baxter, I., Mehlich, M.: Preprocessor conditional removal by simple partial evaluation. In: Proceedings of the Conference on Reverse Engineering. pp. 281–290. IEEE (2001)

[6] Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Science of Computer Programming 52(1-3), 53–100 (2004)

[7] Borba, P., Teixeira, L., Gheyi, R.: A theory of software product line refinement. In: Proceedings of the International Colloquium Conference on Theoretical Aspects of Computing. pp. 15–43. Springer (2010)

[8] Creswell, J.W., Clark, V.L.P.: Designing and Conducting Mixed Methods Research. SAGE Publications (2011)

[9] Easterbrook, S., Singer, J., Storey, M.A., Damian, D.: Selecting Empirical Methods for Software Engineering Research, pp. 285–311. Springer London (2008)

[10] Ernst, M., Badros, G., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering 28(12), 1146–1170 (2002)

[11] Ferreira, F., Borba, P., Soares, G., Gheyi, R.: Making software product line evolution safer. In: Proceedings of the Brazilian Symposium on Software Components, Architectures and Reuse. pp. 21–30. IEEE (2012)

[12] Garrido, A., Johnson, R.: Challenges of refactoring C programs. In: Proceedings of International Workshop on Principles of Software Evolution. pp. 6–14. ACM (2002)

[13] Garrido, A., Johnson, R.: Refactoring C with conditional compilation. In: Proceedings of the International Conference on Automated Software Engineering. pp. 323–326. IEEE (2003)

[14] Garrido, A., Johnson, R.: Analyzing multiple configurations of a C program. In: Proceedings of the International Conference on Software Maintenance. pp. 379–388. IEEE (2005)

[15] Garrido, A., Johnson, R.E.: Embracing the C preprocessor during refactoring. Journal of Software: Evolution and Process 25(12), 1285–1304 (2013)

[16] Gazzillo, P., Grimm, R.: SuperC: parsing all of C by taming the preprocessor. In: Proceedings of the International Conference on Programming Language Design and Implementation. pp. 323–334. ACM (2012)

[17] Gousios, G.: The ghtorrent dataset and tool suite. In: Proceedings of the Working Conference on Mining Software Repositories. pp. 233–236. IEEE Press (2013)

[18] Jackson, D., Schechter, I., Shlyakhter, I.: Alcoa: the alloy constraint analyzer. In: Proceedings of the International Conference on Software Engineering. pp. 730–733. ACM (2000)

[19] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)

[20] Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: International Conference on Software Engineering (ICSE). ACM (2008)

[21] Kästner, C., Apel, S., Kuhlemann, M.: A model of refactoring physically and virtually separated features. In: Proceedings of Conference on Generative Programming and Component Engineering. pp. 157–166. ACM (2009)

[22] Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Transactions on Software Engineering and Methodology 21(3), 14:1–14:39 (2012)

[23] Kästner, C., Giarrusso, P., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of ACM SIGPLAN Object-Oriented Programming Systems Languages and Applications. pp. 805–824. ACM (2011)

[24] Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of International Conference on Software Engineering. pp. 105–114. ICSE, ACM (2010)

[25] Liebig, J., Janker, A., Garbe, F., Apel, S., Lengauer, C.: Morpheus: Variability-aware refactoring in the wild. In: Proceedings of the International Conference on Soft. Eng. pp. 380–391. IEEE (2015)

[26] Liebig, J., Kästner, C., Apel, S.: Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In: Proceedings of the International Conference on Aspect-Oriented Software Development. pp. 191–202. ACM (2011)

[27] Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C.: Scalable analysis of variable software. In: Proceedings of the European Soft. Eng. Conference and the Symposium on the Foundations of Software Engineering. pp. 81–91. ACM (2013)

[28] Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., Schröder-Preikschat, W.: A quantitative analysis of aspects in the eCos kernel. In: Proceedings of the European Conference on Computer Systems. pp. 191–204. ACM (2006)

[29] McCloskey, B., Brewer, E.: Astec: A new approach to refactoring C. SIGSOFT Software Engineering Notes 30(5), 21–30 (2005)

[30] Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., Gheyi, R.: The love/hate relationship with the C preprocessor: An interview study. In: Proceedings of the European Conference on Object-Oriented Programming. pp. 999–1022. Schloss Dagstuhl (2015)

[31] Medeiros, F., Ribeiro, M., Gheyi, R.: Investigating preprocessor-based syntax errors. In: Proceedings of International Conference on Generative Programming: Concepts & Experiences. pp. 75–84. ACM (2013)

[32] Medeiros, F., Ribeiro, M., Gheyi, R., Fonseca, B.: A catalogue of refactorings to remove incomplete annotations. Journal of Universal Computer Science 20(5), 746–771 (2014)

[33] Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., Gheyi, R.: An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In: Proceedings of the International Conference on Generative Programming: Concepts & Experiences. pp. 35–44. ACM (2015)

[34] Mongiovi, M., Mendes, G., Gheyi, R., Soares, G., Ribeiro, M.: Scaling testing of refactoring engines. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. pp. 371–380. IEEE (2014)

[35] Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)

[36] Padioleau, Y.: Parsing C/C++ code without pre-processing. In: Compiler Construction, pp. 109–125. Springer (2009)

[37] Platoff, M., Wagner, M., Camaratta, J.: An integrated program representation and toolkit for the maintenance of C programs. In: Proceedings of the International Conference on Software Maintenance. pp. 129–137. IEEE (1991)

[38] Ribeiro, M., Queiroz, F., Borba, P., Tolêdo, T., Brabrand, C., Soares, S.: On the impact of feature dependencies when maintaining preprocessor-based software product lines. In: Proceedings of Generative Programming and Component Engineering. GPCE, ACM (2011)

[39] Schaefer, M., de Moor, O.: Specifying and implementing refactorings. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 286–301. ACM (2010)

[40] Schäfer, M., Ekman, T., de Moor, O.: Challenge proposal: Verification of refactorings. In: Proceedings of the Workshop on Programming Languages Meets Program Verification. pp. 67–72 (2008)

[41] Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Transactions on Software Engineering 39(2), 147–162 (2013)

[42] Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Software 27(4), 52–57 (2010)

[43] Somé, S., Lethbridge, T.: Parsing minimization when extracting information from code in the presence of conditional. In: Proceedings of International Workshop on Program Comprehension. pp. 118–127. IEEE (1998)

[44] Spencer, H., Collyer, G.: Ifdef considered harmful, or portability experience with C news. In: Proceedings of the USENIX Annual Technical Conference. pp. 185–197. USENIX Association (1992)

[45] Spinellis, D.: Global analysis and transformations in preprocessed languages. IEEE Transactions on Software Engineering 29(11), 1019–1030 (2003)

[46] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys 47(1), 6:1–6:45 (2014)

[47] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming pp. 70–85 (2012)

[48] Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: Proceedings of the International Conference on Generative programming and component engineering. pp. 191–200. ACM (2006)

[49] Vittek, M.: Refactoring browser with preprocessor. In: Proceedings of the European Conference on Software Maintenance and Reengineering. pp. 101–110. IEEE (2003)

[50] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. ACM (2011)

Flávio Medeiros is a professor in the Federal Institute of Alagoas, Brazil. His research interests include configurable systems with a high amount of variability, refactoring and software product lines. He received his Doctoral degree in Computer Science from the Federal University of Campina Grande, Brazil, in 2016.

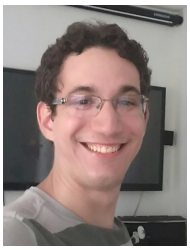
Márcio Ribeiro is a professor in the Computing Institute at Federal University of Alagoas. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco (2012). He also holds the ACM SIGPLAN John Vlissides Award (2010). His PhD thesis has been awarded as the best in Computer Science of Brazil in 2012. In 2014, Márcio Ribeiro was the General Chair of the most important scientific event in Software of Brazil, the Brazilian Conference on Software (CBSoft). His research interests include configurable systems, variability-aware analysis, refactoring, empirical software engineering, and software testing.

Rohit Gheyi is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods, and software product lines. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco.

Sven Apel is the leader of the Software Product-Line Group funded by the esteemed Emmy Noether Programme of the German Research Foundation (DFG). The group re- sides at the University of Passau, Germany. Dr. Apel received his Ph. D. in Computer Science in 2007 from the University of Magdeburg, Germany. His research interests include novel programming paradigms, software engineering and product lines, and formal and empirical methods. He is the author or coauthor of over a hundred peer-reviewed scientific publications. Sven Apel has been a program committee member of several highly ranked international conferences. His work received awards by the Ernst Denert Foundation and the Karin Witte Foundation.

Christian Kästner is an assistant professor in the School of Computer Science at Carnegie Mellon University. He received his PhD in 2010 from the University of Magdeburg, Germany, for his work on virtual separation of concerns. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.

Bruno Ferreira holds a Master degree in Computer Science from the Federal University of Alagoas. His research interests include product lines, software engineering and assistive technology.



Luiz Carvalho is a student of Computer Science at Federal University of Alagoas. His research interests include refactorings, formal methods, and mutation testing.

Baldoino Fonseca is a professor in the Computing Institute at Federal University of Alagoas. His research interests include refactoring and data analysis. He holds a Doctoral degree in Informatics from PUC-Rio.

APPENDIX A
THE C MODEL

In this appendix, we present the model of a subset of the C language that we used to generate programs with application possibilities for our refactorings automatically, as discussed in Section 7.1. The subset that we consider includes local and global variables, function definitions, `if` statements, and the following types: `char`, `int`, and `float`. We have not considered pointers, structures, loops, and concurrency.

Based on our C model, we used the *Alloy Analyzer* [18] to find instances that satisfy the model constraints. By using the instances provided by the *Alloy Analyzer*, our tool *Colligens* converts the instances into real C configurable programs with application possibilities for our refactorings. *Colligens* is responsible to introduce preprocessor conditional directives, such as `#ifdef` and `#endif`, in the generated programs. We have not considered the C preprocessor language in our model because of the complexities of dealing with undisciplined directives. As we discuss in Section 2, undisciplined directives can appear anywhere in the code and may wrap only parts of C constructors, making their specification in *Alloy* difficult.

In Listing 1, we present part of the C model in which we define signatures to represent the main structure of a C program. We define that C program is a translation unit signature that contains a set of declarations. We define an identifier signature to name variables and functions that need to have unique identification. Next, we define a variable that has a specific type. We have signatures for other elements, such as statements, local and global variables, and parameters. Our complete C model is available at the Web site of the project.[12]

Listing 1: Declarations of the C model.

```
abstract sig Declaration {}
sig TranslationUnit {
  declares: set Declaration
}
abstract sig Identifier {}
abstract sig Variable {
  type: one Type
}
// more signatures...
```

In Listing 2, we present a signature for function definitions. In C, a function is a declaration with a unique identifier that receives a set of parameters, returns a value, and contains a set of statements. Notice that we considered in our model only functions that receives a single parameter. When considering functions with multiple parameters, *Alloy* caused an explosion of states, and we could not generate valid programs. Furthermore, all functions considered in our model must return a value and must have exactly one if statement in its body. The reason to add these constraints is to generate programs with application possibilities for our refactorings.

12. http://fmmspg.appspot.com/refactorings/index.html

Listing 2: Declaration of a C function.

```
sig Function extends Declaration {
  id: one FunctionId,
  returnType: one Type,
  ...
  if: one If,
  returnStmt: lone ReturnStmt
}
```

A valid C program must satisfy a number of well-formed rules. For example, a program cannot have two variables with the same identifier in the same scope, and a function should not have statements after returning a value and finish its execution. In Listing 3, we present a few rules defined in our model. As we can see, we define that all programs must have declarations, all identifier used are unique, and that all local variable are declared in the function body.

Listing 3: Well-formed rules for a C program.

```
fact Rules {
  translationUnitNotEmpty
  allIdentifiersAreUnique
  allLocalVariablesExistInFunction
  // more rules..
}
pred translationUnitNotEmpty {
  all src:TranslationUnit |
    #src.declares > 0
}
// more predicates..
```

To reduce the number of instances generated by the *Alloy Analyzer* with the purpose of avoiding the explosion of spaces, we defined some optimizations, such as that functions cannot have empty bodies, and all programs must have one global variable, one `if` statement, and two function definitions. We present part of the optimization predicate in Listing 4.

Listing 4: Optimizations to avoid explosion of spaces.

```
pred optimization[] {
  ...
  all f:Function | #f.stmt < 4 and #f.stmt > 0
  #Function = 2
  #GlobalVarDecl = 1
  #If = 1
}
```

By using the C model that we specified in *Alloy*, we can generate configurable programs with application possibilities for the refactorings of our catalogue, as we discussed in Section 7.1. After generating the configurable programs, *Colligens* generates the corresponding test cases and applies our refactorings automatically.

In addition to the configurable program presented in Section 7.1, we present another example of generated C program. In Figure 17 (a), we present a configurable program with application possibility for Refactoring 3, with

alternative `if` statements. Notice that the generated program follows the constraints defined in the model, e.g., all functions have a `return` statement and start with a local variable definition. Furthermore, the program does not have functions with empty bodies, contains an `if` statement, and exactly two function definitions. Notice that the generated program can be configured by defining macro `TAG` or not. So, we have two configurations in this program: (1) macro `TAG` enabled, and (2) macro `TAG` disabled. In Figure 17 (b), we present the code that *Colligens* generates after refactoring the source code of the generated configurable program.

```
float Glob = -1.0F;

int Func1(int P0){
  int Local0 = 2;
  return Local0;
}

int Func0(int P0){
  int Local0 = 2;
#ifdef TAG
  if (Glob && Func1(P0)){
#else
  if (Glob){
#endif
    Glob += 1.0F;
    return Local0;
  }
  return Local0;
}
```

(a)

```
float Glob = -1.0F;

int Func1(int P0){
  int Local0 = 2;
  return Local0;
}

int Func0(int P0){
  int Local0 = 2;
  bool test;
#ifdef TAG
  test = Glob && Func1(P0);
#else
  test = Glob;
#endif
  if (test){
    Glob += 1.0F;
    return Local0;
  }
  return Local0;
}
```

(b)

Fig. 17: Example of generated program with an application possibility for Refactoring 3.