FeatureCoPP: Unfolding Preprocessor Variability

Kai Ludwig
Harz University of Applied Sciences
Wernigerode, Germany
kludwig@hs-harz.de

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

ABSTRACT

Annotation-based and composition-based variability mechanisms have complementary strengths regarding software maintenance and evolution. Consequently, several proposals have been made to combine, integrate, and substitute both mechanisms. An open challenge is to provide a unified, automatic, and practical technique to adopt such proposals. In this paper, we present a technique to convert variable feature code that is enclosed in the C preprocessor's conditional compilation into compositional feature modules and vice versa. We facilitate the usability of our technique by keeping the annotation-based representation of the C preprocessor. Besides contributing a practicable implementation, we describe the core principles of our technique and demonstrate its functionality based on previous empirical studies and by analyzing the Linux kernel. While our technique is fast in transforming projects, we also illustrate the challenges of maintaining fine-grained feature modules.

CCS CONCEPTS

• **Software and its engineering → Preprocessors**; **Software product lines**; *Feature interaction*; *Maintaining software.*

KEYWORDS

Software product lines; preprocessor; variability analysis; empirical study; software metrics

ACM Reference Format:
Kai Ludwig, Jacob Krüger, and Thomas Leich. 2020. FeatureCoPP: Unfolding Preprocessor Variability. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20), February 5–7, 2020, Magdeburg, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377039

1 INTRODUCTION

Variability is an important concept to enable systemic reuse and customizing of software variants. In particular, variability is often managed as a software product line, which builds on systematically reusing *features* [1, 29]. Such features represent a user-visible characteristic of a system and can be optional or mandatory [13]. While mandatory features are used in all variants of the system, optional features can be selected and deselected, and thus are present only in a subset of all variants.

In order to implement optional features using variation points, several *variability mechanisms* can be used [1, 10]. Such mechanisms can be categorized depending on their representation, either as annotation-based—using annotations in the code base and removing unselected features—or composition-based—separating features into modules and integrating selected ones into the code base. There are arguably complementary pros and cons to these representations concerning program comprehension and maintenance, with partly contradicting empirical evidence [1, 8, 17–19, 32, 35]. Consequently, several researchers argue to combine both representations or transform code from one to the other [3, 14, 16, 20, 21].

To facilitate the practical adoption of such ideas, we proposed to *integrate* both representations, enabling a preprocessor to handle both [21]. In this paper, we present a corresponding solution by introducing a technique and tool that can automatically extract optional features implemented with conditional directives (CDs) of the C preprocessor (CPP)—the most widely used variability mechanism in practice [7, 11, 25]—into feature modules. Our tool, the *Feature COmpositional PreProcessor (FeatureCoPP)* can be applied with different options to perform a complete or selective feature extraction. FeatureCoPP implements a merge function to reintegrate modules into the code base. In detail, our contributions are:

- We describe a technique to automatically separate CPP variability into feature modules.
- We report an evaluation of our technique, including its performance, verification, and discussion of its usability.
- We share our tool as an open-source project on GitHub.[1]

The results show that our technique allows to decompose CPP variability quickly. Still, there are some program limitations, such as the structure of some modules potentially harming maintenance and comprehension. These problems result from the granularity, discipline, and expressions allowed by the CPP [19, 25, 26].

2 COMPOSITIONAL ANNOTATIONS

In this section, we motivate our technique based on related and previous work. Then, we describe the concept of compositional annotations (CAs) [21] as basis for our tool.

2.1 Motivation

Several authors have proposed to combine composition-based and annotation-based representations. This way, developers can select the representation that is most suitable for them. First ideas to combine and transform representations have been proposed by Kästner and Apel [14], who discuss the pros and cons, and later developed a transformation model for the simplified Featherweight

[1] https://github.com/ldwxlnx/FeatureCoPP

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377039

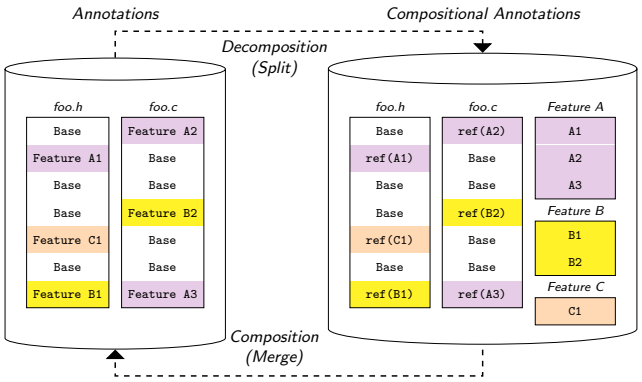


Figure 1: Concept of compositional annotations.

Java [16]. Based on this idea, case studies have been conducted that indicate the practical problems of combining two variability mechanisms [4, 20]. A more advanced technique to tackle this issue is projectional editing [3, 28], which still has to prove its practical applicability. These works indicate the usability, but also problems, of combining both representations. From such experiences, we proposed to integrate a compositional layer into an annotative one, and discussed the specific requirements and goals [21]. Moreover, in a user study [19], we described this concept to experienced CPP developers. The results show that, depending on feature characteristics, developers see practical applications for our technique—partly for implementing, but especially for analyzing and correcting source code that belongs to a specific feature (e.g., collecting all feature code on demand to understand a bug).

2.2 Compositional Annotations

Analogously to OOP or FOP, where objects contribute to features in different roles [2, 22, 30, 34], in CPP systems CDs act as roles, which contribute to features spread across multiple source files. The idea of CAs is to allow preprocessor variability to comprise separate feature modules (cf. Figure 1). So, instead of having all features cluttered into a single code base, references to parts of a feature module are included in the code [21]. In order to facilitate the adoption of our tool and its refactoring, we require two functions: First, we provide a functionality to automatically extract (*split*) code from CDs into modules, including the creation of suitable references and updating the project structure. Second, we provide the inverse functionality of our preprocessor, allowing to reintegrate (*merge*) these modules back into the code by replacing references with the corresponding code. The textual analysis of our technique aligns to the workflow of the CPP and its corresponding C standard [12].

Decomposition (Split). On the left side of Figure 1, we depict an exemplary input. Here, we consider two files (`foo.h`, `foo.c`) with three optional (`A`, `B`, and `C`) and one mandatory feature (`Base`). The features may be scattered throughout the code base, consisting of multiple variation points (e.g., `A1`), which we refer to as roles. For the CPP, every role is marked with CDs (e.g., `#ifdef`). FeatureCoPP extracts these roles into separate feature modules, resulting in the structure that we display on the right side of Figure 1. Now, three new files exist, each representing one feature and comprising all of its roles. As said, the original position of each role is annotated with a reference that identifies each role in the feature modules.

Composition (Merge). At this point, developers can work with the separated modules, which provide a consolidated view on the feature code. Still, before compiling the system, it must be reassembled. To this end, FeatureCoPP implements a merge functionality that reintegrates the separated modules into the code base. This functionality reversely substitutes all references in the code with the corresponding roles. So, it creates a single code base comprising all features. Logically, if the developers did not employ any change to the feature modules, splitting and merging result in exact copies.

Annotation ≠ Feature. Pohl et al. [29] describe that features can represent *internal* and *external* variability, where internal variability is not exposed to the customer. Furthermore, not all CDs contribute to an actual feature. For instance, include guard CDs only avoid erroneous repeated header inclusions. Consequently, a selective decomposition of features should be possible. For now, FeatureCoPP allows developers to specify feature and file names to extract only a selected subset of variability.

2.3 Project and File Structure

As large-scale software systems can have identically named files, we have to preserve their structure during decomposition.

Project Structure. We differentiate between four entities while analyzing a system:

(1) **Selected files:** A selected file is explicitly selected by a user (i.e., based on its name or a suffix). There are two situations that can appear if a selected file is analyzed: First, the file has user-specified feature code, which is extracted by FeatureCoPP—we refer to such entities as *selected files with feature*. Second, the file does not have user-specified feature code, and therefore is ignored by FeatureCoPP—we refer to such entities as *selected files without feature*.
(2) **Deselected files:** A deselected file is a file the user wants FeatureCoPP to ignore. FeatureCoPP neither inspects nor transforms such files into a CA output.
(3) **Folders:** Folders are only preserved if they contain at least one *selected file with feature* anywhere in their structure.
(4) **Feature modules:** FeatureCoPP composes feature modules from all selected roles that it identifies in the aforementioned selected files. Feature modules have a unique identifier and are stored separately in an additional directory.

File Structure. In Figure 2, we show how FeatureCoPP decomposes *selected files with feature* into modules. On the left, we show the annotation-based input file `src.c`. The outermost code contributes to the mandatory feature `BASE`. `BASE` surrounds a CD `#ifdef A` with code that is a role for feature `A`. Moreover, within this CD, three other roles are nested: `#ifdef B`, `#ifdef A`, and `#ifdef C`.

Our technique transforms the file `src.c` into the CA structure on the right side of Figure 2. FeatureCoPP preserves CDs and extracts only the corresponding variable C code. The C code is substituted with a textual reference, which links to the extracted code in the feature module. We preserve the nesting structure of variability, which results in roles pointing to roles in another (e.g., `Feature A Role 1` → `Feature B Role 1`) or the same (e.g., `Feature A Role 1` →

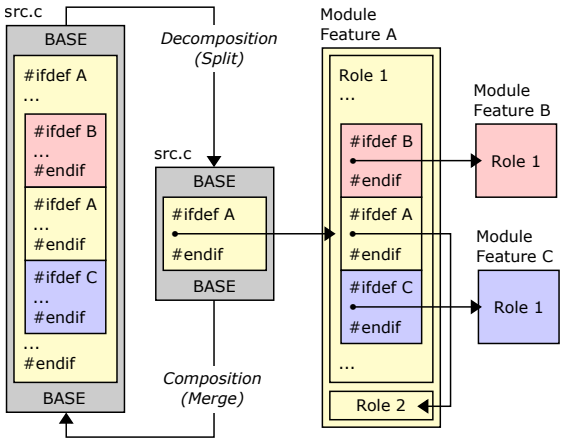


Figure 2: Conceptual overview of FeatureCoPP on file level.

`Feature A Role 2`) feature module. Within feature modules, we use markup text (i.e., annotations) to distinguish roles.

If the user does not request features `B` and `C` to be extracted, the respective feature-related code would remain as is within the CDs. This may prevent further indirections, but potentially clutters a feature with unrelated code. As we keep the original CDs in conjunction with modularized variability, our current implementation can be considered as a hybrid of the inter approach for variant-preserving mapping following the taxonomy of Fenske et al. [9].

2.4 Feature Mapping

Mapping a role to its corresponding feature module creates the problem of making a feature uniquely identifiable. We exemplify this problem with the following equivalence relations:

```
#ifdef A    ≡  #if defined(A)     ≡  #if defined A
#ifndef A   ≡  #if ! defined(A)   ≡  #if ! defined A
```

Each row shows semantically equivalent CDs that differ syntactically. Simply decomposing based on comparing macro names would assign `#ifndef A` to feature `A`, although it is an absence condition, meaning that it does not contribute code to feature `A` [26]. Moreover, `#if` needs to be followed by a constant expression (e.g., macro name, arithmetic, and relational expressions) [12]. Thus, solely using the macro name as identifier for a feature is insufficient. To address these problems and to provide a unified naming, we identify each feature by its constant expression. In case of `#if[n]def`s, we create equivalent constant expressions using `defined` and negation operators. Consequently, we treat explicit feature interactions as one distinct feature. However, this results in the assignment of, for example, `#if A && B` and `#if B && A` to different features, namely `A&&B` and `B&&C`. Moreover, we do not perform constant expression evaluation and macro expansion, meaning that semantically equivalent, but syntactically different CDs, are also assigned to different features. We will address these aspects in future work.

3 DECOMPOSING DIRECTIVES

In this section, we detail the decomposition functionality (split) of our technique that extracts CDs into feature modules.

3.1 Preconditions

First, we have to ensure that the created CA system contains only files and folders that represent the most recent state of the input and aligns to the user's feature selection. To exemplify this requirement, we highlight two scenarios, which potentially impair project integrity and code maintenance:

- A previous execution decomposed all available roles of the code base. When a new selective decomposition is invoked, the new output retains files of the previous execution. Thus, the CA system could become inconsistent with files that do not belong to the new selection.
- Between two executions, the file and/or folder structure of the input project is changed. In this case, files could co-exist as duplicates within different folders, also resulting in an inconsistent CA system.

To avoid these problems, we recursively delete the previous CA system before performing a new decomposition.

3.2 Handling Conditional Directives

We implemented a functionality to detect CDs within the input project. To implement the recognition of CDs directly at the character level, we use the Java-based scanner generator JFlex[2] to create an appropriate platform-independent lexer. During the lexical analysis, we explicitly recognize the following classes of reserved words:

(1) **Conditional Directives:** The CPP keywords `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` are recognized with regular expressions conforming to the C specification [12].
(2) **Comments:** Multi-line (`/**/`) and single-line comments (`//`) are recognized, as they may contain reserved words.
(3) **Line Extension:** We concatenate multi-line CDs (`'\'`) to complete their constant expressions.
(4) **String and Character Literals:** We handle such literals separately, because their content may contain reserved words.
(5) **Anything Else:** We preserve any character input not matching the aforementioned ones as is. This includes disregarded CPP directives (e.g., `#define`, `#include`), C source code, and white space symbols.

Parsing Constant Expressions. To identify roles and features by name (cf. Section 2), we analyze the constant expression of each CD with a Java-based LALR(1) [6] parser we generated with JCup.[3] During this analysis, we also transform `#if[n]def`s to their equivalent counterparts with constant expressions (i.e., `[!] defined`).

Parsing `#else`-Directives. While `#else`-directives do not have macro names or constant expressions, they may contain feature code. To identify roles from `#else`-directives by name, we transform `#else`-directives to equivalent constant expressions by assembling their preceding CDs [26].

3.3 Variant-Preserving Mapping

Having defined the prerequisites to identify roles and features, we now present our technique from the perspective of a variant-preserving mapping [9]. Technically, our extraction maps one source file to $1 + n$ output files (i.e., the transformed file plus feature modules). While it is mandatory to transfer the transformed files with

[2] http://jflex.de/
[3] http://www2.cs.tum.edu/projects/cup/

Listing 1: Example Source for Variant-preserving mapping.

```
1  #if A
2   int a00;    // valid
3  #elif B
4   int b00;    // valid
5  # if A
6     int a10; // dead
7  # elif B
8     int b10; // valid
9  # endif
10  int b01;
11 #else
12  int c00;    // valid
13 #endif
```

Listing 2: Extracted feature module B.

```
1  /*@inline occ_id="4" encl_occ_id="2" src="/tmp/test_split/
       ___FeatureCoPP_modules/2.fcp" directive="#elif B"*/
2     int b10; // valid
3  /*@end*/
4  /*@inline occ_id="2" encl_occ_id="0" src="/tmp/test_split/
       impl_test.c" directive="#elif B"*/
5   int b00;    // valid
6  # if A
7  /*$inline occ_id="3" encl_occ_id="2" dst="/tmp/test_split/
       ___FeatureCoPP_modules/1.fcp"*/
8  # elif B
9  /*$inline occ_id="4" encl_occ_id="2" dst="/tmp/test_split/
       ___FeatureCoPP_modules/2.fcp"*/
10 # endif
11  int b01;
12 /*@end*/
```

identical names and relative folder locations to the output project, this becomes more difficult for feature modules. Ideally, each module would be named according to its feature name (constant expression). However, since different file systems prohibit the use of certain characters, this is not possible for a portable solution. For instance, the operator symbols <, >, ?:, /, |, and * are prohibited within file and directory names on NTFS. To solve this problem in a portable way, we successively assign each feature module a unique number $n \in \mathbb{N}^+$, which simultaneously acts as its filename. For instance, the first feature module we create is named `1.fcp`.

To explain our variant-preserving mapping more thoroughly, we use the exemplary input source file we depict in Listing 1. This file consists of partially nested CDs plus their respective controlled code. Starting from line 1, three CDs span three roles at top-level, namely `#if A` in line 1, `#elif B` in line 3, and `#else` in line 11. Each of these CDs starts a new text scope, which is terminated by the following directive or by an `#endif` (cf. lines 9 and 13). Any kind of text within such a scope is a role that contributes to its feature. For instance, the first role contains an integer declaration in line 2 and contributes to feature A. The integer declaration in line 6 also contributes to feature A, but as a different role. With regards to our example in Listing 1, we emphasize the following important implications for our technique:

(1) The text of a CD itself (e.g., `#if A`, but not its controlled code) contributes to the mandatory base, if it is at top-level—or to an optional feature, if it is nested. For instance, line 1 needs to be written to the source file, since it is at top-level. In contrast, the C code (`int a00;`), all whitespaces, and the comment (`// valid`) belong to feature module A. The nested `#if A` in line 5 is part of feature module B, while its controlled code in line 6 belongs to feature A.
(2) Our technique performs a flat extraction of the controlled code and disregards file inclusion (`#include <file>`). Thus, every file is inspected in isolation and without inlining further source files.
(3) We do not expand function or object macros, and thus disregard the unsatisfiability of CDs. This means, that roles within CDs, such as `#if 0` or the nested `#if A` in line 5 of our example, are extracted if their feature name is selected by the user, although they are not processed by the CPP.

After the preconditions (cf. Section 3.1) are met, our technique for a variant-preserving mapping performs a depth-first search to locate the selected source files. These files are scanned for CDs that match the selected constant expressions. The textual structure of the located CDs is preserved by setting up a two-dimensional stack. In the first dimension, we preserve the nesting structure of CDs, while we store the ordering of sibling directives in the second dimension. At the bottom of the stack, we always insert the mandatory base feature (i.e., the transformed file). Each bucket in the stack is connected to the appropriate feature module. By writing the currently processed text on top of the stack, we map the respective source code to the appropriate feature module. We write the text of CDs to the previous top-most bucket. This can either be the source file or another feature module if the CD is nested. In case a feature has not been selected, we traverse the stack to its bottom, either finding a selected feature or the base feature.

With regards to the I/O strategy, our technique works on a per-file level. After an inspected input file is analyzed, we systematically cache all of its textual content and write it to the respective output files—the transformed file and feature modules—during the systematic tear-down of the stack. Thus, we never keep more than the textual volume of the currently processed input file plus additional data structures in heap memory. An alternative strategy could be a complete assembly of the extracted text in heap memory, which arguably can become problematic in case of very large source projects. For instance, the Linux kernel with nearly 500 MiB 8-Bit plain text in all C header and implementations files would require at least a doubling of the text size in memory, due to internal charset conversion to UTF-16 for Java's string representations. Such a drastic growth can quickly result in heap memory shortage for systems with limited resources.

3.4 Output

After the previously described transformation of an input system, our technique generates the corresponding CA system. This output system contains all extracted features in their modules and mappings to all affected source files. We exemplify a simple feature module that is extracted from Listing 1 in Listing 2.

Roles. We enclose roles with markups (i.e., `/*@inline ...*/` and `/*@end*/`) for separation. The markups are embedded in C multi-line comments to avoid lexical inferences with potential tool support (e.g., syntax highlighting, the CPP). Within such a markup, our technique writes the content of the corresponding role. For instance, line 2 of Listing 2 refers to the role comprising the extracted integer declaration from line 8 in Listing 1. As we also extract whitespaces and comments, we emphasize the non-invasive behavior of our technique. Furthermore, each markup contains additional machine and human readable information regarding a particular role. We can

Listing 3: Transformed source file `src.c`.

```
1 #if A
2 /*$inline occ_id="1" encl_occ_id="0" dst="/tmp/test_split/
      ___FeatureCoPP_modules/1.fcp"*/
3 #elif B
4 /*$inline occ_id="2" encl_occ_id="0" dst="/tmp/test_split/
      ___FeatureCoPP_modules/2.fcp"*/
5 #else
6 /*$inline occ_id="5" encl_occ_id="0" dst="/tmp/test_split/
      ___FeatureCoPP_modules/3.fcp"*/
7 #endif
```

identify each role internally based on a unique occurrence identifier $n \in \mathbb{N}^+$, namely `occ_id`. A role could be nested in an enclosing role, which we indicate by adding `encl_occ_id`. Additionally, the file from which this role is referenced, is listed as `src`, together with the CD's text, which encloses the role. This information is required for the composition of a project of CAs, and it helps developers to keep track of the textual relations between particular roles. While not all the information may be necessary to reintegrate feature modules, it supports the comprehensibility of the transformed system.

References. We show a second type of markup in lines 7 and 9 of Listing 2. While the CDs `#if A` and `#elif B` are part of this role themselves, each of them references a further role (i.e., their respective controlled code). Such a referencing is introduced by the markup `/*$inline ...*/`. Here, `occ_id` and `encl_occ_id` refer to the same meaning, while the `dst`-markup refers to the feature module where the corresponding role is written to. As we show in Listing 2, a role in a feature module can reference another role within that same feature module. The reference in line 9 points to role number 4 (`occ_id`) within the same file (`2.fcp`). This particular role is located at the beginning of the file (lines 1 to 3). With regard to Listing 1 (cf. lines 3 and 7) this makes sense, as there is a nested feature `B` at this point. The ordering of the roles within a feature module results from our depth-first writing, including the stack ordering during variant-preserving mapping. While this can seem confusing, it is mitigated by the internal numbering of roles, which helps in tracing back the respective structure.

Source File. Finally, we outline the transformed source file of Listing 1 in Listing 3. A source file is always the root of references in a system of CAs. While the CDs in lines 1, 3, 5 and 7 are preserved, our technique references the respective roles with markup text in lines 2, 4 and 6. From this combination of preserved annotations in conjunction with extracted variability, originates the name *compositional annotations*.

4 REINTEGRATING MODULES

Reintegrating feature modules encompasses the backward transformation from CAs towards a source project of pure annotation-based variability. We have to perform this transformation before an actual translation of the system with standard build mechanisms, such as configure scripts or makefiles.

4.1 Preconditions

The preconditions for a reintegration refer to the prototypical state of our tool. To prove the correctness of our technique, we need an annotation-based output system that we can compare to the annotation-based input system. In case the extraction works correctly, a reassembled system and its original counterpart have to be identical. Since a system of CAs consists only of the user-selected optional and its mandatory feature code, other system components are missing (e.g., invariant sources, documentation, scripts, etc.). Thus, a direct reintegration of an extracted system would only result in reassembled variable source files, which makes a system comparison impossible. To solve this issue, our technique copies the original annotation-based system before an actual reintegration. Afterwards, the copied files are overwritten with feature modules of the CA system. So, an immediate functional composition of extraction and reintegration, without intermediary source modifications will result in two identical systems. For the future, we aim at performing the reintegration directly into the original system, which makes the duplication unnecessary.

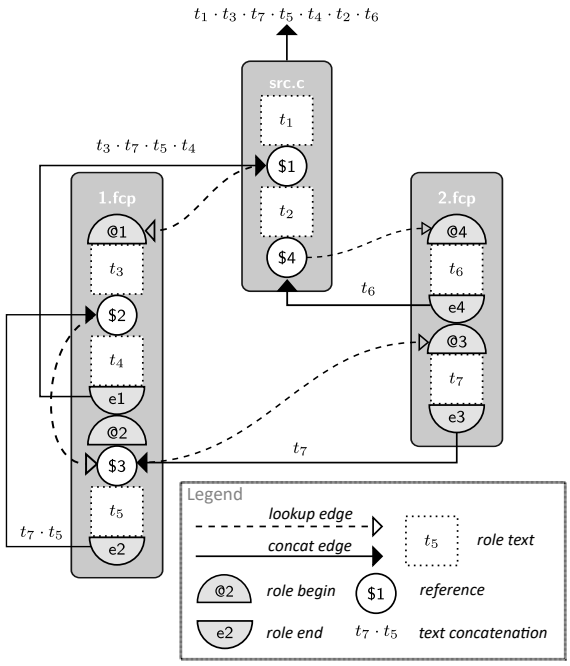


Figure 3: Reintegrating feature modules.

4.2 Depth-First Text Concatenation

We exemplify the process of reintegration more thoroughly in Figure 3. With regards to Listing 1, we have a system of CAs, which consists of one source file (`src.c`) and two feature modules—`1.fcp` for feature `A` and `2.fcp` for feature `B`. We omit the third feature module `! (A || B)`, since it is not required for a basic understanding of the functionality and to improve the clarity of our example. The reintegration starts with localizing all copied source files, namely `src.c` in our example. Our technique now reads the source file line-wise and buffers the text in memory until it identifies the first reference (`$1`). If we detect such a reference, we parse it for the role number (`occ_id`) and the file name of the feature module (`dst="..."`), which we find immediately before reading of the source file continues. We display this with `lookup`-edges.

Afterwards, our tool starts detecting the role `@1`. We parse the corresponding feature module for the role number, which we extract

from the reference. If we find the role, its source code is concatenated to the same text buffer, which we depict as `concat`-edge. Three situations may occur:

(1) The role is read without finding further references, which means that within this particular role no CDs are nested. After appending the last character of the role's source code, we stop processing the feature module.
(2) A reference is detected within the role, which is immediately dereferenced, as described for the source file.
(3) After reaching the end of the feature module file, the searched role is not found. In this case, the integrity of the physically separated project is impaired and the reintegration finishes with an error.

Since role `@2` comprises reference `$3`, the lookup continues in feature module `2.fcp`. Here, we locate and concatenate t_7 to the text buffer before we stop processing this feature module. Within feature module `1.fcp`, we read t_5 and append it to the text buffer after t_7. We propagate this concatenation backwards to where the dereferencing of `$2` started. By this means, we correctly inject the text compound $t_7.t_5$ between t_3 and t_4. The reintegration of source file `src.c` finishes when the end-of-file is reached. At this stage, the source code is completely reassembled in the text buffer and written to the respective output file.

5 STUDY DESIGN

Subject System. To evaluate the applicability of our technique on large-scale real-world software, we conducted a case study using the Linux kernel, which is a well explored [5, 24, 33] and continuously evolving system. The kernel comprises more than 10,000 features implemented with the CPP. Moreover, it includes nearly 15 million lines of source code in 18 thousand header and 24 thousand implementation files. So, we argue that the Linux kernel is appropriate to evaluate our tool.

Research Questions. To investigate the practical usefulness of our technique, we formulate two research questions:

RQ$_1$ *Is the performance and resource consumption of our tool suitable for real-world scenarios?*

RQ$_2$ *What impact do compositional annotations have on software maintainability?*

Methodology. First, we decompose the Linux kernel with our tool, using all header and implementation files. This transformation is performed twice:

(1) **Informed:** We only decompose kernel features, namely CDs containing `CONFIG_` macros.
(2) **Uninformed:** We decompose all CDs without considering naming conventions, extracting any variability; even deliberate dead code (e.g., `#if 0`) or code in include guards.

We use informed and uninformed decomposition of variability as lower and upper bounds for our evaluation. Developers who are familiar with a system will arguably prefer an informed transformation for a defined set of features. We use this scenario as lower bound with regards to performance. In contrast, a complete, uninformed transformation of a system represents the worst-case scenario; so it represents our upper bound with regards to performance. Both scenarios allow us to reason about a real-world application of our tool.

Table 1: Properties of our test system.

CPU:	4x AMD Phenom II X4 945, 800MHz, L1-Cache 512 KB
RAM:	4x 8GB, DDR3, 1333 MHz
OS:	Debian GNU/Linux, 7.11, 64-Bit
Filesystem:	XFS, block size 4096 Bytes
Harddisk:	Samsung SSD 850 EVO 500 GB
Java-VM:	Java HotSpot 64-Bit Server VM (build 25.131-b11)

Second, we perform a reintegration on both, informed and uninformed, transformed systems to show how the respective input relates to the inverse transformation. To address our first research question (**RQ$_1$**), we measure computation and heap usage, and the required time of the transformations. Our evaluation is based on the hardware and software we describe in Table 1.

To perform our tests, we use standard system configurations without optimizations regarding CPU, filesystem, operating system or the Java virtual machine. We measure the duration of decomposition and re-integration effectively within the application, which means that runtime for the preconditions *deletion* and *copying* is not included. We obtained our measurements from single tool executions after assuring comparable system states (e.g., cpu and memory load, process activity) The heap consumption is profiled with VisualVM[4] within Eclipse™ Oxygen.2.

Reflecting on Maintainability. To tackle our second research question (**RQ$_2$**), we reflect on the maintainability of CA projects. To this end, we build on the results of surveying its practical applicability [19] and using it to analyze systems [26]. Moreover, we use the results we obtained during our case study to compare the findings and qualitatively discuss the usability of our tool and CAs.

6 RESULTS & DISCUSSION

In this section we present and discuss the findings of our case study, starting with a summary of the measured data in Section 6.1. Within Section 6.2, we illustrate the applicability of our tool from a technical perspective, analyzing important performance aspects, such as, runtime and resource consumption. In Section 6.3, we reflect on CAs in terms of maintainability. We conclude by showing the current limitations of FeatureCoPP in Section 6.4.

6.1 Results

We report the transformations and their required execution times in Table 2. In the first row, we show the results for an informed transformation—decomposing only the actual Linux kernel features. Analogously, we show the results for an uninformed transformation in the second row—decomposing all CDs, even if they may not represent external variability. For the investigated Linux kernel we decomposed 11,103 kernel features (`CONFIG_`) with 49,996 roles into feature modules (`.fcp`) in the CA system.

For the uninformed transformation, the number of conditional directives that are decomposed into feature modules increases by a factor of three to 36,226. The number of roles is more than doubled compared to before (103,135). This shows the high degree of CDs that do not represent external variability. However, actual features

[4] https://visualvm.github.io/

Table 2: Times for decomposition and re-integration.

Transformation	#Features .fcp	#Roles	Time (Sec.) split	merge
Informed	11,103	49,996	68	181
Uninformed	36,226	103,135	73	456

(CONFIG_) seem to comprise a larger number of roles on average, compared to the remaining CDs.

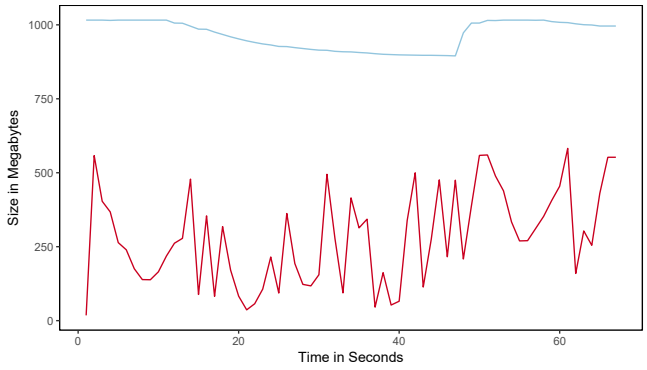
6.2 RQ_1 – Performance

Runtime. While FeatureCoPP decomposes the Linux kernel remarkably fast in the informed transformation (68 seconds), the reintegration takes nearly three times as much with 181 seconds. The reason is the increasing I/O overhead during the reintegration, when FeatureCoPP has to repeatedly open the same feature modules. This effect increases for an uninformed, and thus complete, decomposition of the Linux kernel. Here, reintegration takes unsurprisingly more time (456 seconds), as more feature modules must be processed resulting in more I/O operations. In contrast, for extracting features, FeatureCoPP's execution time increases only slightly to 73 seconds, despite the drastic increase in modules and roles. Remarkably, the analysis of presence conditions (i.e., CDs) with FeatureCoPP is faster and equally reliable compared to existing static type checkers—as these tools do parse the system [23].

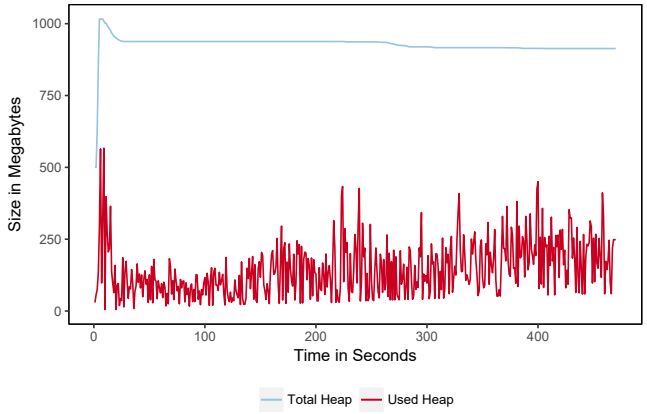
Heap Usage. A potential bottleneck for our Java application is the garbage collection, when a shortage of heap memory occurs. To investigate the memory consumption, we monitored both transformations to explore how the runtime is affected by memory management. In Figure 4, we display the results for the uninformed transformation (the worst case), but the graphs for informed are very similar. The red graph represents the heap consumption of the available heap memory of the Java VM—drawn in blue. For both transformations, shortly after starting FeatureCoPP, the VM reallocates from 512 to 1024 MiB to handle the internal data structures. As the graph shows, the reintegration results in more frequent heap usage changes, due to many files being read into objects with short lifespans. So, its memory consumption is actually better than for the extraction. Overall, the results show that the heap usage of FeatureCoPP is—even for the Linux kernel—rather slim and does not run out of memory with normal VM settings.

CPU Usage. Another aspect affecting program runtime is the CPU usage, since a high utilization of the processor by (a single) application(s) may impair system performance. We inspected the CPU usage of FeatureCoPP during the uninformed extraction and reintegration and show the results in Figure 5. The spikes after program invocation illustrate the load produced by the precondition activities (i.e., copying files during the initial depth-first filesystem traversal). Besides occasional spikes above 50%, FeatureCoPP mainly utilizes 25% of the CPU. We also find few garbage collector activities with slim CPU usage. Again, the results show that our tool should pose no problems in its practical usage.

Validation. To validate that our technique works non-invasive for both, extraction and reintegration, we performed a recursive file and directory comparison with the *diff*-tool—comparing the



(a) Heap usage for uninformed extraction.



(b) Heap usage for uninformed reintegration.

Figure 4: Heap usage for uninformed working mode.

original input project and its reintegrated copy (cf. Section 2). Both projects are textually and structurally identical, besides platform-based differences regarding line breaks (\n vs. \r\n), and character encodings in comments, strings, and character literals. This happens if the application is transformed on a different platform than the one on which the input text has been encoded. We tolerate this invasiveness, since it is related to the experimental state of our application and will be improved in future versions.

Summary. All results indicate that our tool can be used in normal Java VMs. Consequently, we argue that the practical applicability of our tool is ensured and may be integrated into other techniques or frameworks. For example, we used FeatureCoPP in a static analysis framework [23], which was unproblematic and worked equally well as comparable tools we integrated.

6.3 RQ_2 – Reflecting on Maintainability

Comprehensibility. Since the CPP allows lexical variability, annotation-based systems allow very fine-grained feature code [1, 15]. This can result in a *scattering and tangling* of features, hampering code comprehension, and thus maintainability [27, 35, 36]. In case of separating fine-grained code into feature modules, this problem increases [19, 21]. Especially, decomposing code on statement level or below results in a removal of the related functional context. Thus, developers will face problems especially during bottom-up

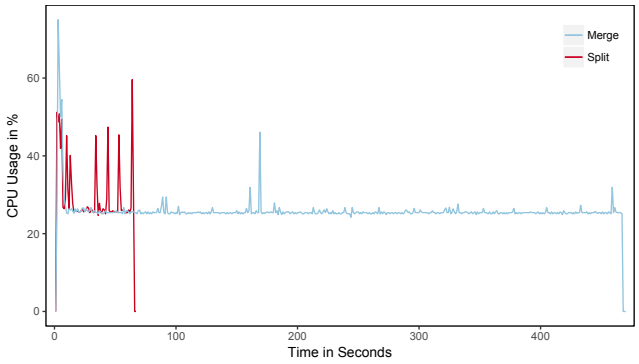


Figure 5: CPU usage for both working modes.

comprehension [38, 39], due to inexplicable execution plans and reduced knowledge of investigated type information of symbols. Such issues have been anticipated by a majority of developers that we surveyed to assess the conceptual approach of CAs [19]. In contrast, a majority of developers found CAs useful in an analysis use case, when, for example, quick feature outlines are requested. Nevertheless, a more sophisticated experiment with developers and our tooling is needed, to investigate comprehensibility aspects more thoroughly, which we will address in future work.

System Integrity. For our technique, the overall system integrity relies on the presence of all feature modules. The loss of one module renders a valid re-integration impossible. Since a feature module can crosscut through a multitude of source files, it is challenging for developers to figure out which files are affected—to which some developers refer to as *action-at-a-distance* [17–19]. To support developers in tracing affected files, we preserve the entire CPP variability for each CA system in an XML structure. Still, there are pros and cons of this representation, but further empirical studies are necessary.

Summary. Overall, FeatureCoPP achieves its goal of decomposing features into modules, which can have pros and cons. Due to the filter capabilities we implemented and the user feedback we received, we argue that such transformations can support the analysis, comprehension, and maintenance of software systems. The problems that may occur due to fine-grained features are rather due to the legacy variability we decompose automatically.

6.4 Limitations

Due to the prototypical state, FeatureCoPP faces some limitations.

Tool Support. To support developers, we need additional tooling for FeatureCoPP (e.g., IDE integration). In particular, there can be fine-grained roles that cannot be parsed without their context. For instance, without context it is undecidable if the C code `a * b;` is a multiplication or a pointer declaration. So, we aim to integrate our technique into an IDE that enables real-time analysis of the code to investigate such issues. This way, we can provide developers a tool suite for our technique and also improve its capabilities.

CPP Compliance. At its current state, our tool accepts only macro names that align to usual C identifiers, conforming to the C standard [12]. However, this standard also allows lexical extensions, which are made in recent *gcc* and *clang* implementations. If our tool is applied on systems that use of such extensions (e.g., *gcc* itself), our parser refuses to transform the respective project.

Lexical Inferences. Although it is commonly argued that the CPP is language independent, its capabilities of string concatenation and character disambiguation imply that this is untrue. Consequently, compiler suites (e.g., *gcc*) discourage the application of the CPP on other languages, due to lexical interferences—although they provide switches to make the processing of some languages possible [37]. To this date, our tool only accepts the interaction of the CPP and the C language. For instance, the interaction of the CPP with assembler code, which occurs twice in our version of the Linux kernel, is refused by our parser. To allow developers a means to bypass this limitation, we provide a blacklisting mechanism, to explicitly exclude affected files from processing.

7 THREATS TO VALIDITY

Internal Validity. Our performance analyses (cf. Section 6.2) focuses on the resource consumption of our tool itself. However, the behavior of an application relates to far more confounding variables, such as, the current overall system load, operating system scheduling strategies, and cache management. Although the stabilization of all confounding variables is hardly possible, we endeavored to use homogeneous system states for profiling. Nevertheless, such variables may have impacted our results.

External Validity. Due to the analysis of only one subject system, we cannot draw conclusions on possibly further limitations of our tool when applied to other systems. Furthermore, since our approach is I/O-intensive, the introduced runtime measures will differ on different platforms and filesystems, due to changed journaling and I/O caching strategies [31]. As this paper focuses on the technique itself and its applicability with regards to time behavior and resource consumption, we argue that the Linux kernel provides an appropriate upper bound to gain insights.

8 CONCLUSION

In this paper, we introduced the concrete implementation of compositional annotations (CAs) [21]. Our implementation allows to decompose variable, annotated C code into feature modules and the reverse reintegration of the resulting CAs into an annotation-based project. We demonstrated that our technique extracts the Linux kernel features in around one minute, while the respective reintegration has a runtime of approximately seven minutes, due to the currently I/O intensive implementation. Also, we show that our technique is modest in terms of memory and CPU consumption.

For future research, we aim for case studies with more subject systems and a removal of our tool's limitations. Further improvements and tool integrations will help us to conduct user and empirical studies to assess its usability, as well as the suitability of such code transformations in general. We also plan to improve the analysis capabilities of our tool, as especially this use case has been highlighted as valuable in a developer survey [19].

ACKNOWLEDGMENTS

This research has been supported by the German Research Foundation (DFG) grants LE 3382/2-3 and SA 465/49-3.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines*. Springer.

[2] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371.

[3] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 563–574.

[4] Fabian Benduhn, Reimar Schröter, Kenner Andy, Kruczek Christopher, Thomas Leich, and Gunter Saake. 2016. Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process. In *Conference on Advances and Trends in Software Engineering (SOFTENG)*. IARIA, 102–109.

[5] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. 1999. Linux as a Case Study: Its Extracted Software Architecture. In *International Conference on Software Engineering (ICSE)*. ACM, 555–563.

[6] Frank DeRemer. 1969. *Practical Translators for LR(k) Languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[7] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146–1170.

[8] Wolfram Fenske, Sandro Schulze, and Gunter Saake. 2017. How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness. In *International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, 77–90.

[9] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2014. A Taxonomy of Software Product Line Reengineering. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 4:1–4:8.

[10] Cristina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 26, 3 (2001), 109–117.

[11] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016), 449–482.

[12] ISO/IEC. 2011. *Programming Languages - C*. Technical Report 9899:2011. International Standards Organization.

[13] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis: A Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University.

[14] Christian Kästner and Sven Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*. University of Passau, 35–40.

[15] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *International Conference on Software Engineering (ICSE)*. ACM, 311–320.

[16] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. *ACM SIGPLAN Notices* 45, 2 (2009), 157–166.

[17] Jacob Krüger. 2018. Separation of Concerns: Experiences of the Crowd. In *Symposium on Applied Computing (SAC)*. ACM, 2076–2077.

[18] Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 338–349.

[19] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *Symposium on Applied Computing (SAC)*. ACM, 2042–2049.

[20] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2018. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience* 48, 3 (2018), 402–427.

[21] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 74–84.

[22] Thomas Kühn and Walter Cazzola. 2016. Apples and Oranges: Comparing Top-down and Bottom-up Language Product Lines. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 50–59.

[23] Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. 2018. PCLocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 284–288.

[24] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering (ICSE)*. IEEE, 105–114.

[25] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202.

[26] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 218–230.

[27] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Leibniz International Proceedings in Informatics*. Schloss Dagstuhl.

[28] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *International Conference on Software Engineering (ICSE)*. ACM, 81–84.

[29] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[30] Christian Prehofer. 1997. Feature-oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443.

[31] Drew S. Roselli, Jacob R. Lorch, Thomas E. Anderson, et al. 2000. A Comparison of File System Workloads. In *USENIX Annual Technical Conference (ATEC)*. USENIX, 41–54.

[32] Alcemir R. Santos, Ivan do Carmo Machado, Eduardo S. de Almeida, Janet Siegmund, and Sven Apel. 2019. Comparing the influence of using feature-oriented programming and conditional compilation on comprehending feature-oriented software. *Empirical Software Engineering* 24, 3 (2019), 1226–1258.

[33] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. 2002. Maintainability of the Linux Kernel. *IEE Proceedings-Software* 149, 1 (2002), 18–23.

[34] Lars Schütze and Jeronimo Castrillon. 2017. Analyzing State-of-the-Art Role-Based Programming Languages. In *International Conference on the Art, Science and Engineering of Programming (Programming)*. ACM, 9:1–9:6.

[35] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 17–24.

[36] Henry Spencer and Collyer Geoff. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Conference*. USENIX, 185–198.

[37] Richard Stallman and Zachary Weinberg. 2016. *The C Preprocessor*.

[38] Anneliese von Mayrhauser and Marie Vans. 1994. Comprehension Processes During Large Scale Maintenance. In *International Conference on Software Engineering (ICSE)*. IEEE, 39–48.

[39] Anneliese von Mayrhauser and Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer* 28, 8 (1995), 44–55.