The Journal of Systems and Software 219 (2025) 112255

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Syntax-preserving program slicing for C-based software product lines ☆

Lea Gerling

University of Hildesheim, Institute of Computer Science, Universitätsplatz 1, 31141, Hildesheim, Germany

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.7565855, https://github.com/LPhD/Jess

Keywords:
Software product lines
Static analysis
Program slicing
Software transplantation
C preprocessor

ABSTRACT

Program slicing is a well-established technique for identifying a reduced subset of a program based on predefined criteria, leading to complexity reduction in subsequent activities. Despite extensive study over the past 40 years, slicing techniques for software product lines *(SPLs)* remain notably scarce. The absence of dedicated SPL slicing approaches hinders their efficient analysis and maintenance, limiting the ability to focus only on relevant parts of the SPL. One reason for this deficiency is the complex nature of a common variability implementation: the use of C preprocessor `#ifdef`-annotations within C code. A slicing approach for C-based SPLs must address the intricate interplay between the C code and the functionality introduced by the C preprocessor. Effectively handling these intricacies will unleash the full potential of SPL analysis.

In this paper, we present a novel syntax-preserving program slicing approach for C-based SPLs. Unlike existing methods, our approach enables the computation of program slices through an integrated analysis of both C and CPP code, while preserving the original program syntax (no element of its syntax is disregarded or changed). This preservation ensures that the resulting program slices remain *authentic* subsets of the SPL, making them suitable inputs for variability-aware analyses. Additionally, we demonstrate the practical applicability of these slices in the context of software transplantation, showcasing their potential for facilitating functionality transfer between different program versions. In contrast to existing transplantation approaches, our solution works without test cases, removing the need for product configuration and execution. Consequently, the variability implementation (along with all other contained preprocessor code) is preserved during the transplantation.

We empirically evaluate our approach on four distinct open-source SPLs, showcasing its effectiveness in generating diverse program slices tailored to different slicing criteria. We asses the accuracy of our code representation, the time required for slicing and transplantation, the size reduction achieved through the slices, and the functionality of our variability-aware transplantation approach.

1. Introduction

Context: Software product lines (*SPLs*) play a vital role in hardware-near systems, offering flexibility and quality while minimizing development costs (Linden et al., 2007). A common practice in SPLs involves the use of conditional compilation with C preprocessor *(CPP)* `#ifdef`-annotations to tailor the software for different hardware configurations (Liebig et al., 2010). This often results in projects with ten thousands of `#ifdef`s scattered all over the code base (Tartler et al., 2014; Liebig et al., 2010). Moreover, a software system is often extended asynchronously by different people (Mens, 2002). Consequently, there may exist different variants of the same system in parallel, which partly contain different functionalities. These variants can be created systematically, for example, through configuration as part of an engineered SPL (Linden et al., 2007), or unsystematically, for example, through copying and adapting an existing program (also known as *clone-and-own* Rosiak et al., 2019; Bergey et al., 1998; Dubinsky et al., 2013 or software *forking* Sung et al., 2020; Zhou et al., 2018; Stănciulescu et al., 2015; Bitzer and Schröder, 2006; Ernst et al., 2010). Furthermore, software evolves over time, where the different versions also partly contain different functionalities (Mens, 2002).

Problem: A common task for software developers in this context is sharing or migrating an existing implementation, like, for example, a new functionality or a bug fix, from one version or variant to another (Zhou et al., 2018). The automated migration of improvements among related software variants is still an unsolved problem for C-based SPLs (Gerling and Schmid, 2020). In general, there exist numerous search-based approaches that address this problem for non-variable software, like software transplantation (Barr et al., 2015) or history slicing (Li et al., 2017). Those approaches have in common that they are not variability-aware (ignoring CPP code) and often rely on the

☆ Editor: Professor Laurence Duchien.

E-mail address: gerling@sse.uni-hildesheim.de.

https://doi.org/10.1016/j.jss.2024.112255
Received 25 January 2024; Received in revised form 30 August 2024; Accepted 8 October 2024
Available online 16 October 2024

presence of unit test cases, which are not widely used in the context of C-based SPLs. Thus, a SPL developer must conduct an error-prone and long lasting manual analysis to migrate an existing implementation.

Solution idea: A solution idea for the described problem is the use of *program slicing* to identify the lines of code that implement a desired functionality, which can then be migrated to the target software. Program slicing is a technique introduced by Weiser (1981) that reduces a program to a relevant subset of itself called *program slice*. In its original form, this slice contains all statements that influence a given variable at a given program location. Program slicing relies on the use of static analysis and *system dependence graphs* (Horwitz et al., 1988), which contain information about the control- and data-flow of a program. Therefore, it is not dependent on the existence of test cases. The idea to use program slicing for the automated migration of existing implementations is not new for non-variable software (Horwitz et al., 1989), but to our knowledge there exists no comparable approach for C-based SPLs.

Challenges: The use of program slicing to solve the described problem in the context of C-based SPLs imposes a number of challenges regarding the code representation and the slicing process, which we first described in our earlier work (Gerling and Schmid, 2020, 2019). First, C and CPP statements must be contained in the code representation that serves as a input for the slicing approach. Only if this requirement is fulfilled, elements from both languages can be considered during the slicing process. However, due to the different natures of the two involved languages, there exists no joined language model that can be reused. Second, the variability-structure and -information in the resulting program slice must be preserved to get an exhaustive subset of its original program. This requires a variability-aware slicing process that explicitly considers relations previously unregarded in existing slicing approaches.

Solution approach: Our proposed solution approach addresses the described challenges with a dedicated code representation called variability-aware Code Property Graph (*vaCPG*) and a novel slicing process, whose initial concepts are both presented in a previous idea paper (Gerling and Schmid, 2019). A *CPG* is a joint code representation that contains an AST, a data- and control-flow graph, as well as different property information like statement type, parent file, etc. The idea of CPGs was first introduced by Yamaguchi et al. (2014). We extend the CPG to become variability-aware by including CPP statements and additional analyses. Further more, our novel code representation contains all information required for *syntax-preserving* slicing. This is an extension on top of variability-awareness, as our approach does not only consider CPP statements used for variability implementation (`#ifdefs`), but also all other kinds of CPP statements, like includes or macros. In contrast to other existing variability-aware code representations generated by tools like TypeChef (Kästner et al., 2011), SuperC (Gazzillo and Grimm, 2012), or others (Thüm et al., 2014), and slicing approaches relying on them, our slicing approach computes a program slice containing C and CPP code (including `#ifdef`, `#include`, and `#define` statements), resulting in an exhaustive subset of its original program.

Contributions:

- A novel, variability-aware, and syntax-preserving code representation designed for program slicing (*Section* 3.1).
- A fully automated slicing approach for calculating exhaustive slices of C-based SPLs (*Section* 3.2).
- A practical demonstration of automated software transplantation using slices of C-based SPLs (*Section* 3.3).

2. Background

This section provides an overview of the essential concepts and terminology relevant to SPLs and program slicing, as well as the introduction of a formal notation to describe the presented code representation and slicing algorithm. Finally, we highlight selected difficulties and challenges of slicing C-based software systems.

2.1. Software product lines

SPLs are software systems comprising a common code base shared among all product variants and variable parts (Linden et al., 2007). The variable parts of a SPL define the differences among the possible software products. For instance, in the context of modern cars, the software must support the existence of an engine, but the specific type of engine, such as diesel or electrical, may vary across different car models. As a consequence, the implementation of car software needs to account for this variability to accommodate various car configurations.

A common method to implement variability is the use of conditional compilation with CPP annotations (Liebig et al., 2010). Before the compilation, the CPP resolves conditional compilation statements to include or exclude certain parts of the source code based on configuration options. There are different forms of conditional compilation statements like `#elseif`, `#ifdef` or just `#if`, which are always followed by a condition. We refer to this condition as *configuration option*, because its evaluation determines the configuration of the resulting software product.

Listing 1: contains a C and CPP code example with variability, which we will use thorough the following sections to incrementally demonstrate our approach. The code example consists of preprocessor code from Lines 1 to 5 which implements the conditional compilation of the function-like macro `func(x)`. The definition of the configuration option `ADDITION` decides, which implementation of the macro is used. In Line 7, there is a comment, and from Lines 8 to 10 is the definition of a C function named `calc`. Leveraging configuration options and conditional compilation enables systematic reuse of existing implementations for different product variants.

```
1  #ifdef ADDITION
2  #define func(x) x + 1
3  #else
4  #define func(x) x - 1
5  #endif
6
7  //Calculates something
8  int calc (int value){
9    return func(value);
10 }
```

Listing 1: Running C code example

2.2. Program slicing

Program slicing identifies semantically meaningful decompositions of programs known as program slices (Reps and Turnidge, 1995). A slice is computed based on a graph representation of the source code, which contains information about control- and data-flows of the program. The slicing algorithm identifies all nodes of the graph that can be reached from a selected node by traversing the graph's edges based on the designated rules specific to the slicing algorithm. Therefore, the graph limits the elements that can appear in the slice.

Definition 1. The required input for program slicing is a directed graph, which consists of a set of nodes $\mathbf{N}$ and a set of edges $\mathbf{E}$, where $\mathbf{N}$ represents all source code fragments of the program $\mathbf{P}$, and $\mathbf{E}$ represents the relations among the code fragments $\mathbf{E} \subseteq (\mathbf{N} \times \mathbf{N})$.

Ferrante et al. (1987) proposed to use a *program dependence graph* (PDG) for program slicing, which later became the typical graph used for slicing (Silva, 2012). Fig. 1 shows a simple example to demonstrate how this graph is built. Part (a) of the figure is a C code example which contains a simple control structure and four different variables. Fig. 1(b) illustrates the control flow of the code example with a control-flow graph. It models the execution order of the code statements from Fig. 1(a).

The PDG is able to model both the control and data dependencies in one directed graph G_{pdg} (Ferrante et al., 1987), which contains a finite

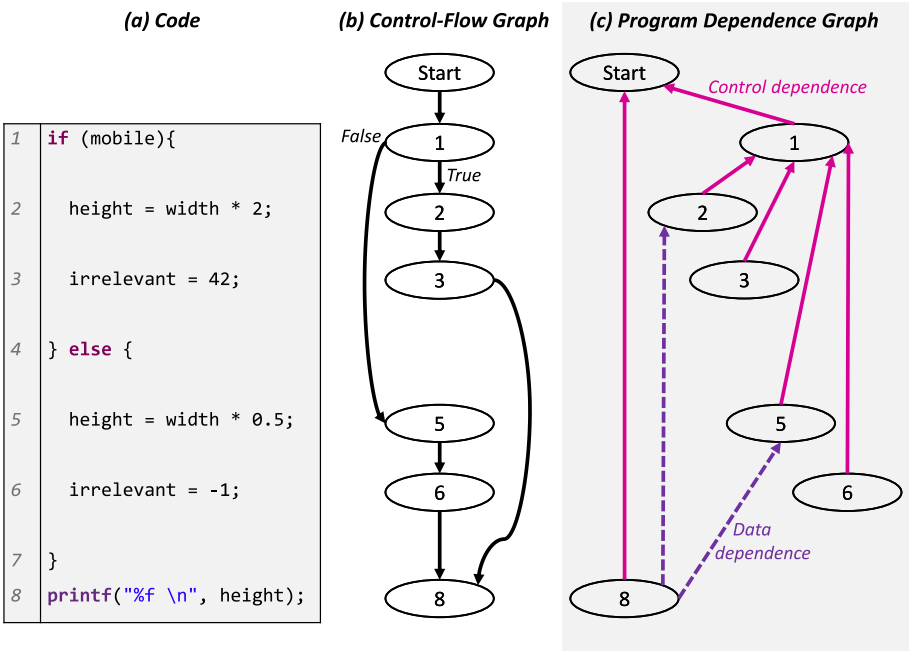

Fig. 1. Code example with its control-flow graph and program dependence graph.

set of nodes N and two ordered sets of edges: *Control dependency* edges E_{cd} and *data dependency* edges E_{dd}, where edge $e_{cd} := (n_1, n_2)$ and edge $e_{dd} := (m_1, m_2)$ each connect a tuple of nodes. Fig. 1(c) shows the program dependence graph for the code example from Fig. 1(a), where the solid lines represent control dependency edges and the dashed lines represent data dependency edges. It starts with a unique Start node and each other node represents one statement from the respective line of the code example on the left.

Given a *control dependency edge* $e_{cd} := (n_1, n_2)$, the execution of n_1 depends on the execution of n_2, and n_2 controls the execution of n_1, meaning if the code of n_2 is not executed, the code of n_1 is not executed either. For example, Node 1 in Fig. 1(c) represents the condition of an if statement which controls the execution of the code from Node 2. When the if statement's condition is false, the code from Node 2 is not executed.

A *data dependency edge* $e_{dd} := (n_1, n_2)$ models the data-flow relations between n_1 and n_2, where the value of a symbol in n_1 can be influenced by the execution of n_2, and the execution of n_2 can influence the value of a symbol in n_1. In other words, n_1 is data dependent on n_2. For example, Node 8 in Fig. 1(c) is influenced by Node 2. Both contain the same symbol `height`, where Node 2 is defining its value and Node 8 is reading its value. Note that the value has not actually to be changed, for example when the same value is set again.

A *program slice* is defined as a subset of program statements that preserve the behavior of the original program with respect to a subset of program variables at a certain location (Weiser, 1981). The selected variables and the location are defined using a *slicing criterion* SC.

Definition 2. A slice of a program P with slicing criterion SC is any syntactically correct program P' that is obtained from P by deleting zero or more statements, so that P' ⊆ P, where the output O(x) for P is equal to the output O'(x) for P', both with respect to SC.

Fig. 2 shows an example slice based on the code from Fig. 1, the original slicing approach from Weiser (1981), and the slicing criterion $\boldsymbol{SC} := (8, \texttt{height})$. The slicing criterion refers to the `printf` statement in Line 8 of Fig. 2(a) and the variable *height*, whose value is printed. Weiser's approach is classified as a *backwards* slicing approach, as it identifies all statements which *influence* the slicing criterion. In the domain of sequential languages, all predecessors of Node 8 are of potential interest, as only their execution can influence Node 8 in terms of control or data dependencies. To identify the predecessor of a node, the control-flow edges must be followed in reverse, therefore leading to the name backward slicing. In contrast, to determine which statements can be *influenced by* a specific statement, the successors of the statement must be identified by traversing the control flow-edges forwards, leading to a *forward* slice (Silva, 2012).

To find more irrelevant nodes other than the successors of Node 8, the control and data dependencies from Fig. 2(c) must be analyzed iteratively. For this purpose, a *reachability* analysis is performed, beginning from the location defined in the slicing criterion. A node n1 is *reachable* from another node n2, if there exist an edge between n1 and n2. Reachability is transitive and can be limited to traversing directed edges in a specific direction, like, for example, a backward slicing approach is limited to following control-flow edges in reverse. The graph in Fig. 2(c) shows that only Node 2 and Node 5 have a data dependency connection to Node 8. Because the execution of both Node 2 and Node 5 depend on the execution of Node 1, this node is also needed in the slice. Nodes 3 and 6, however, have no data relation to variable height and are not needed or related to the execution of any of the other relevant statements. Therefore, these two nodes can be left out of the slice. As a consequence, the final program slice for the criterion $\boldsymbol{SC} := (8, \texttt{height})$ using a backward slicing approach consists of the following nodes $[1, 2, 5, 8]$ from Fig. 2(c), which map back to the lines of code $[1, 2, 4, 5, 7, 8]$ from Fig. 2(a).

Since the first introduction of program slicing, various slicing techniques with different applications have been introduced (Silva, 2012). We refer to program slicing as a static analysis technique based on analyzing source code without executing it. Furthermore, we focus on *static* slicing, where no variable value is restricted (Reps and Yang, 1988; Weiser, 1981).

2.3. Program slicing of C-based software

Program slicing of C-based software systems poses a number of specific challenges caused by the usage of the CPP within the C language. For example, Fig. 3 shows the original C code example from Listing 1: on the left. The code on the right is the result of preprocessing with the definition of the configuration option `ADDITION`. This represents

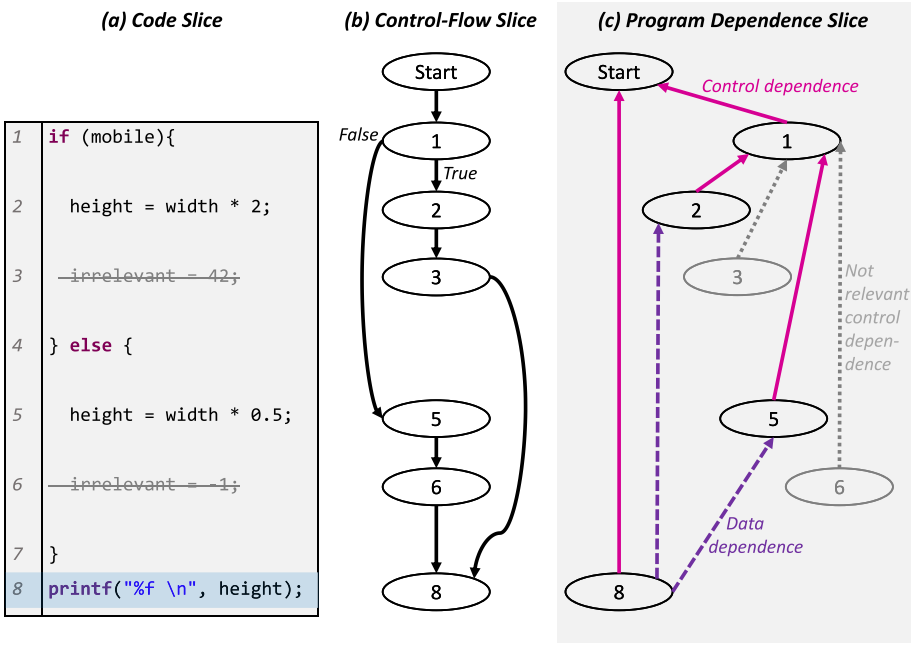

Fig. 2. Slicing example based on Fig. 1.

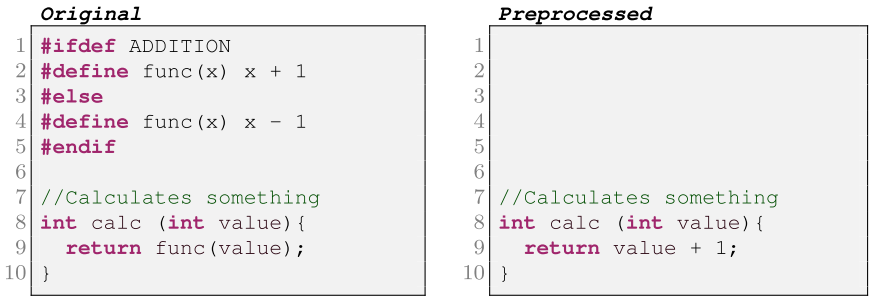

Fig. 3. Running C code example with its preprocessed version.

the input for a non-syntax-preserving slicing approach. When a user is interested in every line of code which can influence the output of the `calc` function, a classical slicing approach would calculate lines 8–10 as result. While this answer is correct, it is not sufficient from a development perspective. It does not reflect the variability of the actual implementation and the dependency on the `ADDITION` configuration option. Moreover, the location of the calculation is also not within the `calc` function, but within two different macros in line 2 and 4 of the source code. This can cause problems when the implementation should be located, for example, to transplant it into another program or to locate a bug.

As a consequence, the preprocessor code must be considered when calculating the program slice for a program. This is problematic for automated approaches, as CPP code has a different syntax than C code and its intrusive nature makes the computation of control- and data-flow graphs complicated. Moreover, file inclusion via `#include` statements is an integral part of the C language, but these relations are not part of the classical slicing theory. Nevertheless, the preprocessor code is part of the developer's view on the program. It can be used for realizing functionality, structuring the code, file inclusion, or implementing different variants of the software for different hardware configurations. Consequently, the CPP code cannot be ignored for calculating a real subset of the program containing all relevant information from a development perspective.

3. Solution approach

In this section, we present our novel approach for syntax-preserving program slicing for C-based SPLs. We first introduce the concept and creation process of our code representation: the *vaCPG*. This graph serves as the required input for our slicing algorithm, enabling a unified modeling of C and CPP code. Next, we describe the syntax-preserving program slicing approach and highlight its distinctive features compared to traditional slicing approaches. The algorithm operates based on the vaCPG, ensuring that the resulting program slices retain the original syntax. Finally, we highlight our first steps towards automatically transplanting a functionality from one version of a program to another, without the need for test cases, leveraging the slicing technique and the vaCPG.

3.1. Variability-aware code property graph

The transformation of a program's code into a graph is a crucial step in the program slicing process. While the PDG is the typical representation used for program slicing (Horwitz et al., 1988), it lacks the necessary capabilities to represent SPLs and support variability-aware and syntax-preserving program slicing. The PDG consists of generic nodes with control and data dependency relations but cannot jointly represent C and CPP nodes or variability relations. To address these limitations, we realize and extend the previously presented concept

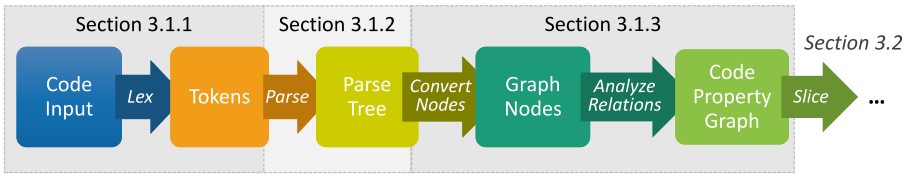

Fig. 4. Code transformation overview.

idea of a vaCPG (Gerling and Schmid, 2019), which overcomes these challenges and encompasses all the required capabilities.

The creation of the vaCPG involves several subsequent steps, as illustrated in Fig. 4. The first two steps overlap with the steps used for building a compiler (Aho et al., 2007): lexical analysis (Section 3.1.1) and parsing (Section 3.1.2). We choose to follow these steps, as they are well-established for creating graphs, including control- and data-flow graphs, from source code, allowing us to leverage existing approaches. Additionally, given the specific requirements of a syntax-preserving slicing approach, we apply further analyses of the created parse tree, including a variability relation analysis. The culmination of these efforts is the *vaCPG creation* (Section 3.1.3), where we combine all the previous results into a single code representation.

3.1.1. Lexical analysis

Our approach begins with a *lexical analysis*, that groups the code input into *lexemes*, which represent the possible words of a language (Aho et al., 2007). These lexemes are determined by a set of rules, which define the content of each lexeme. Instead of implementing a lexical analyzer from scratch, we leverage the existing tool Joern (Yamaguchi et al., 2014), which already contains a rule set for the C language. This approach allows us to focus on the essential task of revising and enhancing the existing C syntax rules with CPP syntax, without being burdened by technical details.

Our contribution lies in conceiving a comprehensive set of lexical analysis rules that form the foundation for syntax-preserving slicing of C-based SPLs.[1] The rule set covers the ISO/IEC 9899:2011 C standard, as well as specific additions from the GNU compiler (Free Software Foundation, 2023), commonly used in the open-source landscape. Additionally, we include rules for comments and newline characters, elements usually skipped in the context of programming languages, as they do not affect the functional content of a program. However, for a fully syntax-preserving slicing approach, no syntactical elements from the original code input should be omitted. The only exception, to some extent, are whitespaces. While we explicitly encode whitespaces where they can optionally appear within a lexeme, we forgo explicit encoding wherever they appear between two other lexemes.

The definition of lexical analysis rules itself is not new; however, what sets our work apart is the creation of an integrated rule set for both C and CPP languages. This unified approach allows us to handle the entire C-based SPL, including the variability annotations, within a single framework. As an example, consider the following lexical analysis rule:

```
PRE_IF: '#' [ \t\u000C]* ('if' | 'ifdef' | 'ifndef');
```

In this rule, ***PRE_IF*** represents the identifier of the resulting lexeme, which matches a hash symbol followed by an arbitrary number of whitespaces, and then either the characters `'if'`, `'ifdef'`, or `'ifndef'`. Such rules ensure that no relevant syntactic elements are overlooked during the slicing process.

[1] The complete rule set can be found at https://github.com/LPhD/Jess/blob/dev/projects/extensions/joern-fuzzyc/src/main/java/antlr/ModuleLex.g4.

3.1.2. Parsing

Parsing is a critical step in our approach, as it transforms the lexemes obtained from the lexical analysis into a *parse tree*, representing the grammatical structure of the input (Aho et al., 2007). Similar to lexical analysis, parsing relies on a set of rules that define the allowed grammar for a language. While the Joern tool already contains a grammar for the C language, we contribute additional parser rules for newlines, comments, and CPP code based on the ISO/IEC 9899:2011 C standard as well as GNU compiler (Free Software Foundation, 2023) specific additions, making it a combined grammar for both C and CPP code in a syntax-preserving way.[2]

The addition of newline rules may seem minor, but it is essential as newlines can appear within statements, serving as a structural element and adhering to project style guidelines. To achieve a fully syntax-preserving approach, we explicitly encode the optional usage of newline characters within our grammar rules. The resulting parse tree is then an authentic model of the code, with no loss of the original visual structure.

Instead of explicitly encoding all newlines, an alternative approach would be to reconstruct the line number from each individual token. However, this would still require the encoding of newlines for rules where a newline defines the ending of a statement, and a way to reconstruct the correct segmentation of multi-line statements. Further, the encoding of newlines within statements has some synergies with our other contributions, which are explained in the next paragraphs. Therefore, we went with this decision.

```
1 #ifdef CONFIG_A //ToDo: Check requirements
2 void
3 #else
4 int
5 #endif
6 function(int *apointer)
```

Listing 2: Variable function definition with comment

Similarly, integrating comments and CPP statements with the C language poses challenges. While CPP code has functional impacts on the C code and implements variability, comments are essential for code understandability and often serve as markers for tasks or potential issues. For example, Listing 2: shows a function definition where the return type of the function is variable and implemented with conditional compilation. Furthermore, Line 1 contains a comment that is integrated inside the function definition. To preserve the interaction of C code, CPP code, and comments, we propose integrated rules for all three elements. For instance, our ***func_def*** rule includes an optional `fragment` rule, which defines rules for comments, newlines, and CPP statements. This enables all three statement types to appear within a function definition, fully preserving the syntax of our example (Listing 2:):

```
func_def: type fragment* identifier fragment*
        func_param_list fragment* '{';
```

[2] The complete rule set can be found in the *.g4 files at https://github.com/LPhD/Jess/blob/dev/projects/extensions/joern-fuzzyc/src/main/java/antlr/.

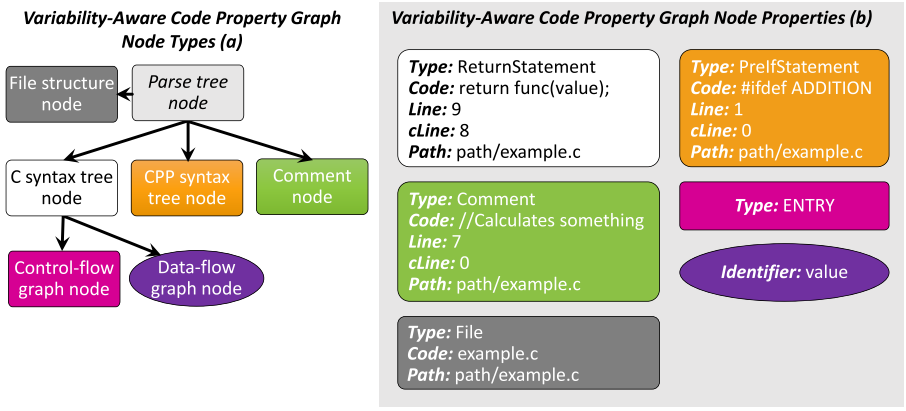

Fig. 5. Variability-aware code property graph node overview.

However, in theory there still might occur situations where the syntax is not detectable by our approach. In this cases, we inform the user about this problem and skip only the problematic input. Based on our experience during the evaluation and research on the usage of the CPP (Liebig et al., 2011; Hunsen et al., 2016; Ernst et al., 2002; Medeiros et al., 2015), such a drastic obfuscation is very rare in practice and is avoided by developers. Further, semantic details, like whether a function is of type `void` or `int`, are not relevant for the conduction of program slicing. Instead, we are only interested in whether there exists a connection between two elements. If this question cannot be answered without fail, we will overestimate and assume that there is a connection.

3.1.3. vaCPG creation

The last step for the transformation of source code into a vaCPG requires a number of additional analyses to construct the attributed nodes and edges of the graph. The input for this step is the parse tree generated based on the grammar described in the previous Section 3.1.2. One of the main differences between a parse tree and a vaCPG is the *attribution* of their nodes. While a parse tree contains only generic nodes that contain the same kind of information, a vaCPG contains different *kinds* of nodes. Fig. 5 shows in the left part (a) an overview of the nodes which are present in our vaCPG and their relation to the *parse tree* nodes, which are not present in the final graph. The original code property graph (Yamaguchi, 2015) contains nodes for C source code, as well as nodes which are the result of control- and data-flow analyses. Further, it contains dedicated nodes to represent the file and folder structure of a program. We extended this collection in our vaCPG by nodes for comments and preprocessor code, which both are derived from the parse tree by matching their parser rule to the respective node type.

Fig. 5 shows in the right part (b) an example node for each node kind present in the vaCPG. Every node has several associated *properties*, which contain attributes of the node, depending on the node kind. These attributes are either needed for program slicing, or for the reconstruction of the original code from the graph. For example, the white top-left node in Fig. 5(b) shows a C syntax tree node example. Its most important properties are:

- **Type**: The associated grammar rule. With this property, we know part of the meaning of a code fragment, for example whether it is a `ReturnStatement` which terminates the control flow of a function. This property is taken from the node type of the parse tree. It is required for the slicing process, as we have dedicated slicing rules for different node types, to make the slicing more efficient.
- **Code**: The associated source code. We changed this property in comparison to the original approach, so that it contains the whole syntax, including punctuations, to allow full syntax preservation. This information is taken from the content of the parse tree nodes. It is required for the slicing process, as well as for the syntax-preservation.
- **Line**: The vertical position of the code fragment in its file, starting at Line 1. For example, top left C syntax node represents the code of Line 9 from the example in Listing 1:. This information is taken from the tokens of the parse tree. It is required for the syntax-preservation to reconstruct the positioning of the code.
- **cLine**: The horizontal position of the first character of the code fragment in its line, starting at character position 0. For example, the first character of the C syntax node is situated at position 8 of Line 9 in the Code example in Listing 1:. This information is taken from the tokens of the parse tree. It is required for the syntax-preservation to reconstruct the positioning of the code.
- **Path**: The relative path of the source file in the program. For example, the nodes in Listing 1: (b) would be situated in a file called `example.c` in a folder called `path`. This information is taken from the tokens of the parse tree. It is required for the syntax-preservation to reconstruct the positioning of the code.

The C syntax node in Fig. 5(b) is a *top-level* node, which means that its code is equal to the original line of code. Top-level nodes may be further broken down into child nodes, which represent different parts of the top-level node. This break-down is achieved based on the grammar rules and similar to the node structure of the parse tree. It is helpful for the slicing process, because complex code fragments can be separated into less complex sub-fragments, which can then be discriminated between relevant and non-relevant fragments for the slicing process.

The orange C preprocessor syntax node in the top-right and the green comment node in the middle-left in Fig. 5(b) have the same properties as the C syntax node. Similar to the C syntax node, they can represent whole lines of code from the original input, or just fragments. All three of these node kinds are directly retrieved from the parse tree, as the bold arrow in Fig. 5(a) illustrates. Their differentiation into the node kinds is done based on their node type, where all nodes matching a `comment` rule are categorized as comment nodes, all nodes matching a rule belonging to the preprocessor language are categorized as preprocessor syntax nodes, and all nodes matching a C language rule are categorized as C syntax nodes.

The gray node in the bottom-left of Fig. 5(b) is a file node. In contrast to the node kinds discussed before, this node does not reference source code, but instead the source file object. Its property code refers to its file name and the path property contains its full path relative to the top-level folder of the project. In addition to file nodes, there are also folder nodes which share the same properties. Folder and

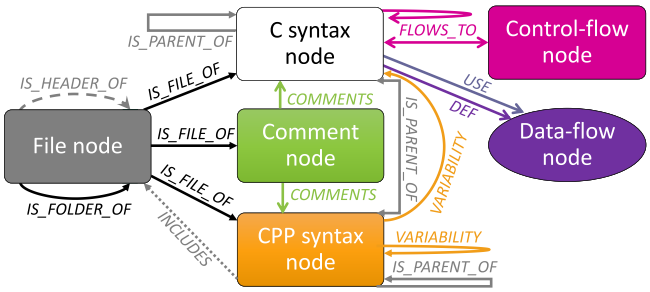

Fig. 6. Meta-model for the variability-aware code property graph.

file nodes are used to represent and reconstruct the file and folder structure of a software project. Further, whole files can be included via the preprocessor `#include` statement, which makes an explicit representation of files necessary.

The remaining two nodes in the right part of Fig. 5(b) are part of the control- and data-flow analysis. The pink middle-right node refers to the entry point into a control flow. It has no other properties besides its type, which can either be ENTRY, for marking the begin of a control flow, or EXIT, for marking the end of a control flow. Entry and exit nodes never appear in the slicing result, they are only needed to form correct control-flow graphs, which always have an entry and at least one exit. The purple bottom-right node with the property identifier is part of the data-flow analysis. Similar to entry and exit nodes, its content does not appear in the slicing result. Instead, the value of the identifier property refers to a variable that is read or written to. Identifier nodes allow to easily identify all data flows from and to a selected variable.

Fig. 6 shows the **complete meta-model for our vaCPG**, including all node and edge types that can be present, and their possible relations. It is a refinement of Fig. 5, as it additionally contains the specific edge types that can occur in the vaCPG. In contrast to parse trees or PDGs, the *edges* in our graph are also attributed with a type property. This allows to distinguish the different connections that are possible among different node types together with their differing meanings. While the original CPG (Yamaguchi et al., 2014) contains edges for C syntax elements (IS_PARENT_OF), control (FLOWS_TO) and data-flow (DEF/USES) relations, as well as file and folder (IS_FILE_OF) relations, we extended the graph to also contain COMMENTS, VARIABILITY, INCLUDES, and IS_HEADER_OF relations based on additional analyses. We will not discuss the previously existing analyses in detail and instead focus on our contributions.

Our new contributions are four additional analyses to create COMMENTS, VARIABILITY, INCLUDES, and IS_HEADER_OF edges. As slicing relies on the identification of relevant connections, the connection of the new node types Comment Element and CPP Syntax Element to the other nodes is crucial. Therefore, we first introduced two new analyses during the conversion process of the parse tree to the CPG: The *comment-analysis* is triggered every time a comment parsing rule is encountered, and the *variability-analysis* is triggered whenever an `#if`, `#ifdef`, `#else`, or `#elseif` parsing rule is encountered.

The comment-analysis checks whether the comment is in the same line as a C or CPP syntax element. If so, the Comment node is connected with the C or CPP Element node with a COMMENTS edge from the Comment node. If not, the next top-level C or CPP Element node after the comment is connected via the COMMENTS edge. However, this methodology is not exact, as C comments have no syntactical mechanism (apart from their positioning) to link them with the code that is meant to be commented. Nevertheless, this relation allows our slicing approach to also retrieve the comments that are probably connected to the source code of the slice, where there sometimes may be even more comments than necessary.

The variability analysis checks whether C or CPP code is enclosed by a conditional compilation block. For this purpose, every line of code that appears after an `#if`, `#ifdef`, `#else`, or `#elseif` statement is connected to this statement with a VARIABILITY connection, until the matching `#endif` statement is reached. Further, `#else`, `#elseif` and `#endif` statements are connected with their opening counterpart via a IS_PARENT_OF edge. Nesting of different conditional compilation blocks is also possible, where the inner blocks are connected with the outer blocks via VARIABILITY edges. As a conditional compilation block per definition has a well-defined beginning and end, the identification of variable code is reliable.

In addition to the two new analyses connecting the new node types, we added two further analyses related to the navigation among files, to allow slicing across multi-file projects: The *includes-* and the *header-analysis*. Both of these analyses involve File nodes and happen after all other analyses are finished. The header-analysis matches *.c and *.h File nodes in the same directory with the same name and connects the respective nodes with an IS_HEADER_OF edge. The includes-analysis matches CPP include statements which include files in the current project directory with the included File node and connects them with an INCLUDES edge. For this purpose, the file name or path used in the include statement is matched with File nodes with the same name in the same directory as the source file of the include statement. If no matching file is found, the search is expanded to the whole directory of the project. External directories are not considered.

Finally, we implemented an automated *accuracy-analysis* for the syntax-preservation capabilities of our vaCPG. Thus, after the transformation from source code to the vaCPG is finished, our approach transforms the graph back to source code and matches the results with the original code. Consequently, the user gets informed about all found differences. While this analysis cannot guarantee the correctness of all properties of the graph, like, for example correct edge or node types, it can guarantee that no syntactical element (except from whitespaces) is lost during the transformation process.

3.2. Syntax-preserving slicing algorithm

The goal of our approach is to calculate a syntax-preserving program slice for a SPL implemented in C. The slicing algorithm operates on the *vaCPG*, which we described in the previous section. This graph represents a program's source code, its decomposition into atomic code fragments, and the relations among those fragments. In contrast to previously existing slicing approaches, we are specifically interested in calculating a *real* subset of a SPL consisting of C and C preprocessor code. This calculation must be done without running the program, as running all configurations of a highly-configurable system is typically infeasible for real world programs (Thüm et al., 2014). Consequently, we describe in this section the conception and realization of an *automated static-analysis-based program slicing approach*, as introduced in our previous work (Gerling and Schmid, 2019, 2020).

Fig. 7 gives an overview of the structure of this section and our slicing approach. Section 3.2.1 describes a ***preAnalysis*** step based on a user-given slicing criterion, which results in an initial number of identified nodes called *entry points* in the vaCPG. These entry points are automatically identified based on the form of the slicing criterion. We explain how the different slicing criteria impact the identification process and the motivation behind them. Section 3.2.2 then explains how the algorithm traverses the vaCPG starting from each entry point identified during the preAnalysis. The algorithm ***iteratively*** identifies additional related nodes based on a set of rules, which are bound to the respective node kind. Section 3.2.3 explains how the initial slice calculated by the previous step is enhanced to the final graph slice. For this process, we implement ***postAnalysis***, which allows for automated fine-tuning based on selected configuration options, as well as an automated inclusion of desired code elements to preserve syntactical correctness.

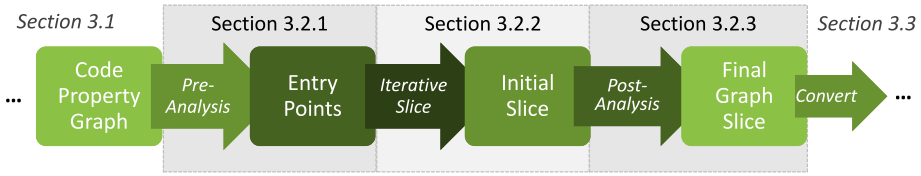

Fig. 7. Slicing overview.

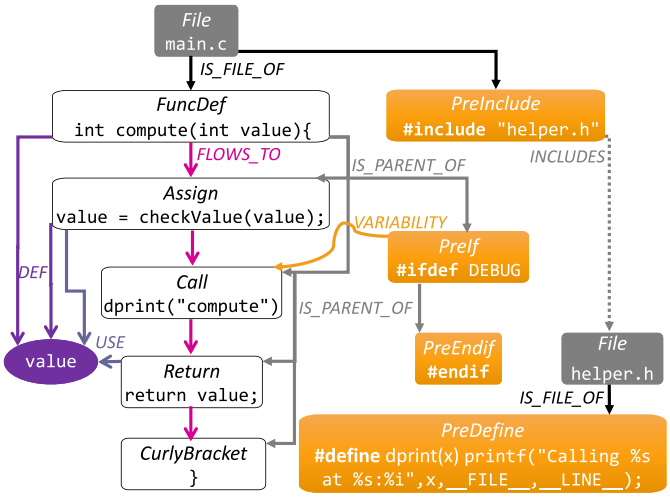

Fig. 8. Simplified vaCPG for the code in Listings 3: and 4:.

3.2.1. PreAnalysis

Algorithm 1 shows the **preAnalysis** step of our slicing approach. In contrast to traditional slicing approaches, which solely implement an iterative reachability analysis, our slicing algorithm utilizes an additional preAnalysis to identify suitable starting points beforehand. This additional step gives the user more control over the slicing result and aids users unfamiliar with the system by automatically identifying relevant locations in the code base. For this purpose, the SC can be given as:

- A *location*, similar to traditional slicing approaches.
- A *configuration option* for special focus on variability.
- An *identifier* that does not require knowledge about the location.
- A *string* that does not require knowledge about locations and has the most open solution space.

Algorithm 1: preAnalysis

Data: Program P, slicing criterion SC
Result: newAnalysisSet
1 **if** SC is location **then**
2 | newAnalysisSet = getStatements(SC);
3 **end**
4 **if** SC is configuration option **then**
5 | newAnalysisSet = getIfDefsAndTheirChildren(SC);
6 **end**
7 **if** SC is identifier **then**
8 | newAnalysisSet = getMatchingIdentifierNodes(SC);
9 **end**
10 **if** SC is string **then**
11 | newAnalysisSet = getMatchingNodes(SC);
12 **end**
13 return newAnalysisSet;

Fig. 8 shows a simplified (all decompositions are missing) example of a vaCPG for the code in Listings 3: and 4: to serve as a running example. Listing 3: contains code from the file `main.c`: An include statement for the header file `helper.h` in line 1 and a function definition for the `compute` function from lines 3 to 9. Inside the `compute` function is a conditional compilation statement, which makes the call in line 6 variable and dependent on the configuration option DEBUG. Listing 4: shows the code from the file `helper.h`: A function-like CPP macro definition named `dprint`, which prints a structured debug output for the parameter `x` on the console. The respective vaCPG contains two gray File nodes for the `main.c` and `helper.h` file, five white C Element nodes for the code in lines 3, 4, 6, 8, and 9 in Listing 3:, four orange CPP Element nodes for the CPP code in lines 1, 5, and 7 in Listing 3:, and line 1 and 2 in Listing 4:, and a purple Data-Flow node for uses and defines of the variable `value`.

```
1 #include "helper.h"
2
3 int compute(int value){
4     value = checkValue(value);
5 #ifdef DEBUG
6     dprint("compute")
7 #endif
8     return value;
9 }
```

Listing 3: Code example: main.c

```
1 #define dprint(x) printf("Calling %s at %s:%
    i",x,__FILE__,__LINE__);
```

Listing 4: Code example: helper.h

The edges in Fig. 8 follow the same notation and logic as described in Fig. 6. The `main.c` *File* node is connected with the *FuncDef* and the *PreInclude* node with an edge, similar to the `helper.h` *File* node which is connected with the *PreDefine* node, as all three nodes are top-level nodes which do not have a parent code node. The remaining C nodes and the *PreIf* node are code children of the function definition due to their nesting, and the *PreEndif* node is child of the *PreIf* node, as it defines the ending of a CPP if statement. The C nodes are connected with FLOWS_TO edges to represent their control-flow, and with USE and DEF edges to the `value` node to represent data flows. The CPP Element nodes are not part of the control- or data-flow analysis. Finally, the *PreIf* node is connected with the *Call* node via a VARIABILITY edge to represent the variable implementation of the call.

The specification of a *location* as SC is meant for users who have knowledge of the source code. They can point at statement or a full line of code, for example line 6 in main.c from Listing 3:, and set this as slicing criterion. The idea is that the slicing algorithm will identify all parts of the input program that are required for the selected statements to fulfill their task, similar to a traditional backward slicing approach. When a location is set as SC, there are no further analyses required and the algorithm will return the node(s) which represent(s) the selected statement(s). Based on the example, this would result in addition of the *Call* node from Fig. 8 to the returned newAnalysisSet.

The specification of a *configuration option* as SC is meant for users who are looking at the code from a variability perspective. It can be seen as a "show me all code that is required to implement this feature". For example, a user could specify DEBUG as slicing criterion.

The algorithm will then identify all conditional compilation statements that reference this configuration option. Based on Listing 3:, this would result in the node representing lines 5 and 7 (the *PreIf* and *PreEndif* nodes in Fig. 8). Finally, the algorithm adds all nodes that are connected with a VARIABILITY edge, as they implement the configurable functionality. This results in the addition of line 6 from Listing 3: (the *Call* node in Fig. 8) to the newAnalysisSet.

The specification of a *string* or *identifier* as SC is meant for users who have no or little knowledge of the source code or the variability implementation. These options can be seen as a generalization of the previous option: Identify all code that is directly implementing this functionality. The user can set an arbitrary string or the name of an identifier as slicing criterion, for example "debug". The algorithm will then identify all code statements or all identifiers that contain this string and add them to the newAnalysisSet. Based on Listing 3:, this would result in the node representing line 5 (the *PreIf* node in Fig. 8). Naturally, this approach only works well when the code statements are named similar to their functionality, or the naming conventions are known to the user.

3.2.2. IterativeAnalysis

After the preAnalysis, our approach starts the **iterativeAnalysis** process shown in Algorithm 2. It starts with iterating over all nodes from the analysisSet (the output of the preAnalysis). For each node, the algorithm identifies all nodes reachable following a given rule set calling the analyzeNode algorithm in line 5. We explain an excerpt of these rules and their general idea in the next paragraphs. The identified nodes are added to the slicing result, as well as to the analysisSet, to determine their reachable nodes then again. During this process, each node is only analyzed once. After all nodes have been analyzed, the iterativeAnalysis process terminates and passes the list of identified nodes to the following postAnalysis step.

Algorithm 2: iterativeAnalysis

```
   Data: Program P, analysisSet
   Result: relevantNodes
1  for node n ∈ analysisSet do
2      if n ∉ checkedNodes then
3          checkedNodes.add(n);
4          relevantNodes.add(n);
5          result = analyzeNode(n);
6          analysisSet.add(result);
7      end
8  end
9  return relevantNodes;
```

The **analyzeNode** algorithm (see Algorithm 3) implements a set of rules to define how a node can be reached from another node. This reachability represents a potential dependency or impact from one node to another, resulting in their required presence in a slice. In traditional slicing approaches, there is only a single rule (like following control flow and data flow edges backwards) and generic nodes. In contrast, our approach works with different node types and implements different rules depending on this type. This enables us to define a customizable[3] rule set for a differentiated treatment of the different node types, making our algorithm more efficient and flexible than a traditional slicing approach that only works with generic nodes. For example, it can skip unnecessary analyses for node types, like CPP endif statements, that cannot extend the slice.

[3] In this paper, we only discuss the standard configuration, more details about the configuration options can be found in the documentation of our tool: https://jess.readthedocs.io/en/dev/slicing.html#configuration-options.

Algorithm 3: analyzeNode

```
   Data: Program P, n ∈ analysisSet
   Result: relevantNodesSet
1  type = getNodeType(n);
2  if type is Call then
3      relevantNodesSet += getMacroOrFunctionDef(n);
4  end
5  if type is FunctionDef then
6      relevantNodesSet += getASTChildren(n);
7  end
   /* More rules for each node type                    */
8  return relevantNodesSet;
```

Further we can write specific analyses for complex cases like *Call* nodes or include a larger number of nodes at once. For example, when n is a *Call* node which results in reaching a *FuncDef* node, we include all child nodes of the called function definition in the slice (as we assume the whole function is needed), instead of individually analyzing the control and data flows in the function. However, this may in rare cases result in the addition of dead (non executable) code to the slice, or the addition of superfluous functionality, when a function is not properly designed and fulfills more than one concern. Hence, we accept that our approach can produce slices that are not minimal and instead always prioritize the completeness of the results. In addition, this approach also allows to include nodes into the slicing process which are missed in traditional slicing approaches, like print statements without a data-flow relation (for the output of user information or status messages), or preprocessor statements.

For example, when the configuration option DEBUG is set as the SC, Algorithm 1 sets the following nodes from Fig. 8 as initial analysisSet: *PreIf*, *PreEndIf*, and *Call*. Then, Algorithm 2 conducts the following iterations and calls to the analyzeNode algorithm:

1. n is the *PreIf* node: No further analyses triggered, relevantNodesSet is empty. **Rationale**: There are no connections to other nodes that have not been explored yet.
2. n is the *PreEndIf* node: No further analyses triggered, relevantNodesSet is empty. **Rationale**: This node type does not have relevant edges.
3. n is the *Call* node: getMacroOrFunctionDef in line 3 of Algorithm 3 is triggered, the *PreInclude* node and the *PreDefine* node are in the relevantNodesSet. **Rationale**: A call can have a definition of the called function or function-like macro either in the same file, or in an included file. The getMacroOrFunctionDef method follows the incoming IS_PARENT_OF and IS_FILE_OF edges until it reaches the parent *File* node. First, it follows the outgoing IS_FILE_OF edges to all existing *FuncDef* or *PreDefine* nodes and looks for a matching identifier. If that fails, it follows outgoing IS_FILE_OF edges to all existing *PreInclude* nodes, and then follows their outgoing INCLUDES edges to the included *File* nodes. For every found *File* node, the algorithm repeats the previous steps, until a definition is found or all reachable nodes have been identified (when the call goes to an external library). If a definition is found, the getMacroOrFunctionDef method returns the definition, as well as all *PreInclude* nodes that were necessary to reach it. The getMacroOrFunctionDef also returns the matching definition in the belonging header or C file.
4. n is the *PreInclude* node: No further analyses triggered, relevantNodesSet is empty. **Rationale**: Additions of connected nodes are only triggered in other analyses that need contents from other files.
5. n is the *PreDefine* node: No further analyses triggered, relevantNodesSet is empty. **Rationale**: There are no connections to other nodes in our example.

3.2.3. PostAnalysis

The iterativeAnalysis from Algorithm 2 is finished when all nodes in the analysisSet have been analyzed once. Finally, our approach conducts the **postAnalysis** (see Algorithm 4) to conduct final enhancements of the program slice. It offers additional options to further refine the result, for example, by adding additional nodes, like comments, or offering options on how to deal with constructs that cannot be reliably analyzed with static analysis, like include statements for external libraries. These analyses are not part of the iterative process, as the affected nodes are outside of the scope of the classical reachability analysis (and could therefore not trigger meaningful subsequent analyses when added to the analysisSet). This algorithm requires the program P and the analysisSet as input. The output is the final program slice P'.

Algorithm 4: postAnalysis

```
   Data: Program P, relevantNodes
   Result: Program slice P'
1  P' = addParentFunctions(relevantNodes);
2  P' = addFilesIncludedBySUFilesRecursively(P');
3  P' = addExternalIncludesForSUFiles(P');
4  P' = addInteralIncludesForSUFiles(P');
5  P' = addGlobalDeclaresForSUFiles(P');
6  P' = addDefinesForSUFiles(P');
7  P' = addVariability(P');
8  P' = addComments(P');
9  return P';
```

The first step of Algorithm 4 is the call of the addParentFunctions method, which ensures that for each node being part of a function's content the belonging functionDef node is also part of the slice. Next, methods called in lines 2–6 handle the addition of statements to the slice whose impact cannot be calculated with static analysis. In particular, the postAnalysis deals with include statements for external libraries, internal files, global declarations of data structures or types, and object-like define statements (string replacements without a parameter). Instead of ignoring these statements, our approach allows the user to individually configure how they are handled, offering options with highest recall (add all the selected statements over all files) or with higher precision but a possibility to miss relevant statements (by adding only those statements present in files currently part of the slice). Finally, the algorithm allows the user to select whether the slice should contain all directly related variability implementations (every PrelF directly connected via a VARIABILITY edge to a node within the slice) and all directly related comments (every Comment directly connected via a COMMENTS edge to a node within the slice).

3.3. Software transplantation

A practical use case for syntax-preserving program slicing is the automated transplantation of functionality between different *versions* (evolved over time) or *variants* (existing in parallel) of a SPL. During development, a software system is often extended asynchronously by different people (Mens, 2002). Consequently, there may exist different variants of the same system in parallel, which partly contain different functionalities. These variants can be created systematically, for example, through configuration as part of an engineered SPL (Linden et al., 2007), or unsystematically, for example, through copying and adapting an existing program (also known as *clone-and-own* Rosiak et al., 2019; Bergey et al., 1998 or software *forking* Sung et al., 2020; Zhou et al., 2018; Stănciulescu et al., 2015). Furthermore, software evolves over time, where the different versions also partly contain different functionalities (Mens, 2002).

Recently introduced automated software transplantation approaches (Harman et al., 2013) allow to transfer a selected functionality from one program (the *donor*) to another (the *target*), saving manual effort for the developers and reducing the risk for human-made errors. However, the existing approaches face a significant drawback: They only work by executing test cases. Therefore, the transplantation results are heavily reliant on the quality and existence of appropriate test cases, which are often absent in C-based software systems (Garousi et al., 2018). Moreover, the dependence on the execution of the software restricts the transplantation to be conducted on *preprocessed* code. Consequently, preprocessor code cannot be transplanted and the involved software systems are restricted to only one *configured* variant each. We research how these disadvantages can be circumvented by using our syntax-preserving slicing approach. Hence, we describe in this section, how our vaCPG and the statically calculated program slices can be used to identify and transplant a functionality from one program to another, without needing test cases or preprocessing.

Fig. 9 gives an overview of the structure of this section and of our transplantation approach, which consists of three subsequent steps: ***conversion***, ***differencing***, and ***merging***. In Section 3.3.1, we explain why and how our vaCPG is leveraged to create a textual representation of the source code enriched with syntax information. For the transplantation process, we convert the identified slice, which contains the functionality to be transplanted, as well as selected parts of the target software to this representation. The textual representation is required for the semi-structural code differencing step, which is explained in Section 3.3.2. In this step, the slice is compared with the target software, to identify similarities and differences between them. In the final step described in Section 3.3.3, the results of the differencing step are used to merge the slice into the target software, effectively transplanting its functionality. The final merging result is outputted as plain source code.

The capabilities of our prototypical approach are tailored to specific use cases and therefore build upon the following **assumptions**:

1. The user lacks precise knowledge of the code that exactly implements the desired functionality.
2. The donor and target versions exhibit some degree of similarity, such as being different versions of the same software.
3. The transplantation process should involve only additions of code to the target software, without requiring removals or changes in the existing code.

Addressing Assumption (1) necessitates the use of a localization technique to identify the implementation of the desired functionality, along with its dependencies. Traditional transplantation approaches typically rely on test cases for this purpose (Barr et al., 2015), but we specifically target software without test cases. As a result, we employ our static-analysis-based syntax-preserving program slicing technique to locate the desired functionality within the donor software. Consequently, the first step of our transplantation approach involves defining a donor, a target, and a slicing criterion.

Assumption (2) initially stemmed from our plan to leverage the Git merge functionality[4] to merge the identified slice into the target software. However, initial tests revealed limitations in using Git merge, as it often produced erroneous outcomes due to its unstructured approach to text comparison. This is a well know limitation for text- and line-based comparison approaches (Apel et al., 2011; Seibt et al., 2022; Buffenbarger, 1995; Mens, 2002; Cavalcanti et al., 2019). Consequently, we realized the need for a specialized merging functionality which considers the *syntactic structure* of C and CPP code. However, the utilized merging functionality still requires equal lines of code in slice and target to identify the correct location for integrating the slice.

Furthermore, our approach is constrained by Assumption (3), which limits the shape of the functionality that can be transplanted. Unlike test-based or search-based approaches, we are currently unable to directly modify the target software. Instead, we add new lines from the slice to the target software while omitting equal lines. Although this limitation presents a clear disadvantage compared to test-based approaches, it also marks the first step towards a novel approach

[4] https://git-scm.com/docs/git-merge.

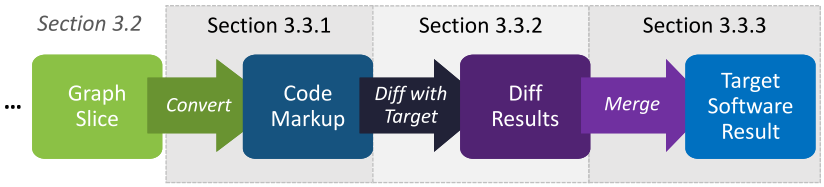

Fig. 9. Software transplantation overview.

Fig. 10. Code markup structure.

for functionality transplantation, eliminating the need for test cases. Furthermore, it offers promising possibilities for future expansion and enhancement.

3.3.1. Code conversion

The first step of our transplantation approach realizes the conversion of the involved code artifacts into a different format. The involved code artifacts are the calculated slice defined by a user-given slicing criterion, as explained in Section 3.2, and the target software. Initially, the slice is available as a vaCPG and the target software is available as source code. As described in Section 3.3, a purely line-based mechanism using the original source code for the transplantation process is not sufficient. We also evaluated the usage of the vaCPG for transplanting the slice into the target software, but graph differencing algorithms are complex and the graph itself contains more information than necessary for a successful transplantation. Moreover, semi-structured differencing approaches seem to be the best compromise regarding effort and precision (Seibt et al., 2022). Consequently, we extract only the necessary structural information from the vaCPG and create a text-based *code markup*. This solution enables the handling of textually equal statements with differing contexts and, thus, prevents false-positive matches. Finally, the markup is designed in a way that it can be easily identified and removed with a regular expression, allowing for a fast transformation back to the original source code.

Fig. 10 shows the structure of our code markup. It combines structural information with the original code in a line-based text format by prefixing the original line of code with an optional *parent function identifier*, *parent block identifier*, or *condition*. The markup is built by traversing an existing vaCPG and extracting all top-level code fragments, effectively restoring the original source files. For all fragments being part of the same line, our approach gathers the related structural information from the vaCPG by analyzing the connected IS_PARENT_OF or VARIABILITY edges. All lines of code belonging to a function are prefixed with the *parent function identifier*, all lines of code belonging to a block, like a multi-line declaration, are prefixed with the *parent block identifier*, and all lines of code being conditionally compiled or executed are prefixed with the corresponding *condition*.

Fig. 11 contains a code example on the left side and its corresponding code markup on the right side. The code markup is simplified to enhance readability, the original markup output contains more characters to make the markup annotations distinguishable from source code. The content and line position of the original source code is not changed, the markup information is appended before the original line. Some lines, like the `#include` statement in Line 1 of the Figure, do not contain additional markup information, as their content is typically unique per file. Other lines, like the `#endif` statement in Line 7 or the curly bracket in Line 9, are now distinctively assignable to their parent construct and cannot be falsely matched to other textually equal fragments of the same kind.

3.3.2. Slice and target differencing

The second step of our transplantation approach involves a comparison of slice and target software to identify different and identical lines of code. Based on the assumptions described in Section 3.3, the goal of this step is locating the parts of the slice non-existent in the target software, to prepare them for transplantation. For this purpose, our approach first compares the source code file names of the slice's artifacts with the target's files. Fig. 12 shows a visualization of this comparison, where the circles represent the set of source files and the arrows represent the derived actions. Files only existent in the slice, as shown in the lower part of the figure, only require a copy of the full file into the target folder structure. A conversion to code markup is not necessary. Files existing in the slice and the target software, visualized by the overlapping circles, require a conversion to code markup, to allow for a semi-structured differencing of the source code. Files only existing in target remain untouched, as shown in the upper part of Fig. 12.

For the code differencing, we decided to simplify the problem, as our transplantation approach is only a first prototype to demonstrate the feasibility of using a fully static analysis-based approach for transplantation. Therefore, our approach relies on a *full equality match* and does not use any heuristics or similarity measures. This means that a line in the slice is considered equal to a line in the target software only if their textual content in the code markup format is identical. By using the code markup as base for comparison, we also compare the structural context of each line, effectively conducting a semi-structured differencing, and therefore preventing false positives, as explained in Section 3.3.1. Furthermore, we only check for lines existing in the slice but not in the target, based on the assumption explained in Section 3.3, making the comparison straightforward. Algorithm 5 shows a summary of our differencing process, which takes the target and the slice as input and returns the files distinctive to the slice in the *newFiles* set and the lines distinctive to the slice in the *codeDiff* set. Furthermore, the algorithm collects the last equal line before the distinctive line, as this is required for our merging process, which is explained in the next Section 3.3.3.

3.3.3. Result merging

The final step of our transplantation approach realizes the *merge* of the additional source code lines from the syntax-preserving program slice into the target software. Together with the previous steps, this process happens fully automated, with no human intervention needed. Initially, we also planned to implement an automated conflict solver, to handle lines with minor differences or removals from the target software. However, as these conflicts may miss a distinctively correct solution, we refrained from creating an approach that would need human intervention to produce reliable outcomes. Nevertheless, problems may occur when both versions differ too much, as outlined in Assumption (2) in Section 3.3.

The algorithmic idea of our merging process is described in Algorithm 6. It requires the results of the previous code differencing step (*newFiles*, *codeDiff*) and the target software (without code markup) as input. Based on the list of completely *newFiles*, the algorithm copies the files exclusive to the slice, as well as potential parent folders, to the top-level target folder. The code merging step, beginning in Line 2 of

Example.c

```
1 #include "example.h"
2
3 int compute(int value){
4     value = value * value;
5 #ifdef DEBUG
6     dprint(value);
7 #endif
8     return value;
9 }
```

Code Markup

```
1 #include "example.h"
2
3 ~intcompute~ int compute(int value){
4 ~intcompute~     value = value * value;
5 ~intcomputeDEBUG~ #ifdef DEBUG
6 ~intcomputeDEBUG~     dprint(value);
7 ~intcomputeDEBUG~ #endif
8 ~intcompute~     return value;
9 ~intcompute~ }
```

Fig. 11. Code example and its markup.

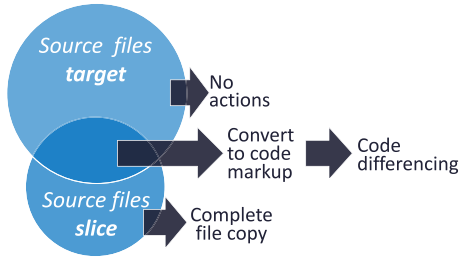

Fig. 12. File differencing and derived actions.

Algorithm 5: Semi-structured Differencing

```
Data: Target, slice
Result: newFiles, codeDiff
for file f ∈ slice do
    if f is ∈ target then
        targetFileContent = createMarkupForTargetFile(f);
        sliceFileContent = createMarkupForSliceFile(f);
        for line lSlice ∈ sliceFileContent do
            if lSlice is ∈ targetFileContent then
                lastEqualLine = lTarget;
            end
            else
                codeDiff.add(lastEqualLine);
                codeDiff.add(lSlice);
            end
        end
    end
    else
        newFiles.add(f);
    end
end
return newFiles, codeDiff;
```

Algorithm 6, consists solely of copying the identified additional lines of code contained in the *codeDiff*, from the slice to the target software. For this purpose, we rely on the information about the last equal line (the *anchor*), which is identified during the differencing step and stored in the *newFiles* list. If there is no anchor, the lines are added at the end of the file. As we build the merged target files dynamically, we can insert the new lines from the slice directly during their identification. The anchor is used to add a new line of code below it and before any other lines distinctive to the target software. This is a design decision to make the merging easier, as we do not need a look-ahead. Alternatively, the new lines could also be added before the next equal line. Finally, our merging approach also removes the existing code markup required for the differencing. It produces plain C and C preprocessor source code, preserving the syntax of the transplanted slice, as well as the syntax of the target.

Algorithm 6: Transplant Merging

```
Data: Target, newFiles, codeDiff
Result: mergedTarget
mergedTarget.createNewFiles(newFiles);
for line l ∈ codeDiff do
    if l is lastEqualLine then
        insertPosition = getFileAndLineFromTarget(l);
    end
    else
        mergedTarget.insertNewPlainLinesFromSlice(insertPosition,
         l);
    end
end
return mergedTarget;
```

Fig. 13 shows an example result for a calculated slice on the left and the respective target on the right. The lines from the slice are added in the same parent file in the target software and keep their relative order. The last equal line before a line distinctive to the slice is marked in purple (slice and target). It serves as an anchor point to determine the exact position of the line insertion from the slice. In the example, Line 4 is the anchor for the new lines 5–7 from the slice. As a consequence, the old Line 5 of the target code is now moved to Line 8, with all subsequent lines following the same adjustment.

4. Evaluation

The main goal of our evaluation is to evaluate the functionality of our novel code representation and syntax-preserving program slicing approach, as well as the usability of the calculated slices for software transplantation. For this purpose, we conduct a real-world benchmarking study to assess the accuracy of the created vaCPG and the slicing results in different transplantation scenarios. Further, we measure standard performance metrics of time consumption and the size of the resulting slices. However, we do not aim at a generalizable performance analysis, a comparison to other non-syntax-preserving slicing approaches, an empirical analysis of the slice's attributes, like the work presented in Binkley and Harman (2004), or a comparison to fully developed test-based transplantation approaches. Instead, we answer the following research questions:

- **RQ1** How much of the source code is accurately represented in the vaCPG?
- **RQ2** How much size reduction can be achieved by the calculated slices?
- **RQ3** How much time is required to calculate syntax-preserving slices for realistic software systems?
- **RQ4** How often is it possible to automatically transplant a functionality strictly relying on static analysis?

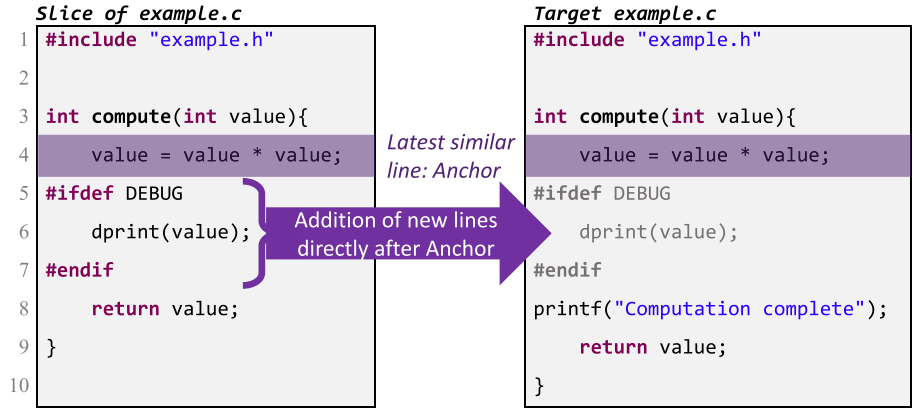

Fig. 13. Merge result example.

4.1. Methodology

This section describes how we evaluate our syntax-preserving program slicing approach for C-based SPLs in relation to our research questions. With **RQ1** we evaluate the **accuracy** (*percentage of code accurately modeled in the graph*) of the vaCPG, which is crucial for the correctness of the slicing result. For this purpose, we use the built-in accuracy-analysis of our vaCPG creation process, as described in Section 3.1. After transforming a program to the vaCPG, the analysis transforms the graph back to source code, matches it with the original code using the git diff (Conservancy, 2023) functionality, and documents any identified disparities. This rigorous analysis ensures that the vaCPG's accuracy is validated and guarantees that any errors or discrepancies in the graph would be faithfully reflected in the slicing results. Therefore, an accurate vaCPG is fundamental to the accuracy and reliability of our slicing approach.

With **RQ2** and **RQ3**, we investigate the practical applicability of our slicing approach. Time consumption is a critical factor in software projects, as time is a limited resource. Manual analysis of complex software systems can be time-consuming, and thus, we are interested in evaluating the time-saving potential of our approach. To achieve this, we measure the **size reduction** (*relative size of the slice in comparison to the original program's size*) achieved by our slicing approach, as a smaller analysis object takes less time than analyzing the entire software. Additionally, we measure the **time required** (*duration in seconds*) for the automated analysis to determine its feasibility in a realistic scenario where developers cannot afford to wait days for the analysis results.

With **RQ4**, we evaluate the functionality of our approach in the context of software transplantation. Since our approach is novel and has fundamentally different functionality and requirements compared to existing approaches, we do not directly compare it to other functionally distinct transplantation methods. Instead, we empirically assess **how often** (*number of successful transplantation scenarios*) our approach can correctly identify a slice and transplant it to the target software. The correctness of a slice is determined based on its slicing criterion, as introduced in Definition 2 from Section 2. To assess whether the target software correctly computes the outputs for the slice, we utilize test cases. To minimize bias and errors stemming from misunderstandings of the target software, we refrain from writing the test cases ourselves. Instead, we select software systems with existing test cases for our evaluation.

4.1.1. Case selection

To identify suitable input programs for our evaluation, we use the following criteria:

1. Our goal is to assess the practical applicability of our approach, therefore we use real world open source projects written in mostly C and CPP for our evaluation. We settled with GitHub (GitHub, 2023) as our source, because it is widely used and contains millions of projects.
2. The project needs automated test cases to validate the correctness of the slice (**RQ4**), as explained in the previous Section 4.1. Note that this criterion excludes a majority of prominent C-based SPLs, like the Linux kernel (Organization, 2023) or BusyBox (Erik Andersen, 2023).
3. Based on the previous criterion, we require the program to be automatically installable and executable on Ubuntu 18, as this is a requirement of our tool.
4. We look for programs that are actually used, by sorting the results based on popularity (GitHub uses the measurements of stars for this).

Based on these four criteria, we selected the following software projects for our evaluation: *scrcpy* (Vimont, 2023), *Silver Searcher* (Greer, 2023), the GNU *grep* (Foundation, 2023) implementation, and the *PHP interpreter* (Group, 2023). *Scrcpy* is a tool that allows users to mirror the screen of an Android mobile phone on a computer and consists of approximately 6000 logical lines of code (LLOC) (note that the size of projects may change over time). *Silver Searcher* and *grep* implement a search functionality for code and both have around 4000 LLOC. Finally, the *PHP interpreter* is responsible for executing the scripting language PHP. We focused on the main implementation without libraries, which consists of about 28,000 LLOC. These projects cover a range of different functionalities and sizes, providing a diverse set of input programs for our evaluation.

The next step in answering **RQ4** involves identifying suitable donor and target versions of the software, while adhering to the assumptions detailed in Section 3.3. To achieve this, we conducted an analysis of the version history for each project, aiming to pinpoint commits that introduce significant new functionality (more than 3 additional LLOC) composed exclusively of additional lines of C and CPP code, along with an accompanying test case. For this analysis, we utilized an automated commit analysis tool (LPhD, 2023) to initially identify a set of potential fitting commits. Subsequently, a manual analysis was carried out, taking into consideration the semantics of the changed code, the test case, and the commit messages. This manual scrutiny enabled us to precisely identify a suitable slicing criterion and assess whether the slice contains, at minimum, the functionality introduced in the chosen commit. By utilizing the associated test case, we could further validate whether the slice produced the desired outputs.

Due to variations in development practices, the results of this process varied across projects. For example, in the relatively new and small *scrcpy* project, which typically included test cases alongside new functionalities, we found a large number of fitting scenarios (163 from automated analysis) and stopped searching for more scenarios after identifying 8 that met our criteria through manual analysis. In contrast, the older and larger *PHP* project did not follow a similar approach,

Table 1
Evaluation scenarios.

Scenario	Donor	Target	Slicing criterion
scrcpy:1	f9d2d99166f2259f05cd348fcde019b9a314d780	ec71a3f66ab48c2fdd1728753acc09edbd4db570	*Location:* File: app/tests/test_device_event_deserialize.c, Line: 7, Type: FunctionDef
scrcpy:2	2a3a9d4ea91f9b79b87387a4883af484e702c53a	ca0031cbde8cc4aaed07ddb7523841754c0423d5	*Location:* File: app/tests/test_strutil .c, Line: 190, Type: FunctionDef
scrcpy:3	12a3bb25d39037931f577dc0165b3e517aad92df	3ee9560ece4b4d17031e23b3ac8a0901f86c5694	*Location:* File: app/tests/test_control_msg_serialize.c, Line: 216, Type: FunctionDef
scrcpy:4	1982bc439b67829ec815b5e1bd13f1bf4e660f4b	ef56cc6ff74862e85c0af4746fa26ef391f6f3a9	*Location:* File: app/tests/test_cli.c, Line: 36, Type: FunctionDef
scrcpy:5	1982bc439b67829ec815b5e1bd13f1bf4e660f4b	ef56cc6ff74862e85c0af4746fa26ef391f6f3a9	*String:* lock_video_orientation
scrcpy:6	eb0f339271862af38693913e5ce1d4078d2c9e56	bdd05b4a16a2b0fc15d2ecf8f832afb9670854ee	*Location:* File: app/tests/test_control_msg_serialize.c, Line: 239, Type: FunctionDef
scrcpy:7	eb0f339271862af38693913e5ce1d4078d2c9e56	bdd05b4a16a2b0fc15d2ecf8f832afb9670854ee	*String:* rotate_device
scrcpy:8	63c078ee6ca19c4a2b3d84b04068220ef425aaf4	3149e2cf4a9d7b0db34c30d41077a73f048cb919	*Location:* File: app/tests/test_control_event_serialize.c, Line: 181, Type: FunctionDef
SilverS.:1	73999f1e59272f80fbe79d33d0d619d74211f678	7b24ee5d22d8c43581c21303875723ea6b30c095	*Identifier:* passthrough
SilverS.:2	73999f1e59272f80fbe79d33d0d619d74211f678	7b24ee5d22d8c43581c21303875723ea6b30c095	*String:* ‘‘passthrough’’
SilverS.:3	33d9d711766cbf3c5d9b52aa471722522b231d94	abd982483eabb91b7ccf9eed2a918f638c149a7d	*Identifiers:* [invert_regexes, invert_ regexes_len]
SilverS.:4	33d9d711766cbf3c5d9b52aa471722522b231d94	abd982483eabb91b7ccf9eed2a918f638c149a7d	*Strings:* [‘‘invert_regexes’’, ‘‘invert_ regexes_len’’]
SilverS.:5	33d9d711766cbf3c5d9b52aa471722522b231d94	abd982483eabb91b7ccf9eed2a918f638c149a7d	*Location:* File: src/search.c, Line: 481, Type: FunctionDef
SilverS.:6	c0e7e35be4662672046e1f98aa11c66c786af79e	a744f26ead18fb4c4e6727fbb701b9e5d73cf1c2	*Identifier:* matches_len
SilverS.:7	c0e7e35be4662672046e1f98aa11c66c786af79e	a744f26ead18fb4c4e6727fbb701b9e5d73cf1c2	*Location:* File: src/util.c, Line: 126, Type: FunctionDef
SilverS.:8	210a8763e0b550de9615b890dd1c18e9aa265486	81e22e2756c203cf7df3ee6dd78a77b036e945b7	*Identifier:* print_path
SilverS.:9	210a8763e0b550de9615b890dd1c18e9aa265486	81e22e2756c203cf7df3ee6dd78a77b036e945b7	*Location:* File: src/options.c, Line: 427, Type: ExpressionStatement
grep:1	51ef8adb2f7eeb073ba98be4f6baf56817e4d358	6ada8b0a1b3f408ebf76e0cf7f7bb61019a70fdc	*Location:* File: src/grep.c, Line: 2414, Type: FunctionDef
grep:2	d5bfcc2daa123fa0e8660909052f7ca2ec6b7649	11ce80861109361570cbedda6a966264367f7c76	*Location:* File: src/main.c, Line: 1883, Type: FunctionDef
grep:3	8f365f028427e4f8f6f52e548a3a0cbf3c9e94db	72a4a19a115b0227dd31dc87ed9c4d428eaae9e2	*Configuration Option:* HAVE_LANGINFO_ CODESET
grep:4	83a95bd8c8561875b948cadd417c653dbe7ef2e2	d433dad3f4183eba763f73345055ba4800cdef43	*Location:* File: src/kwset.c, Line: 588, Type: FunctionDef
php:1	99c48a24779dab32e004b2bc99441155e217c08f	9a744c66e70d34293eb3726cc9d17a3ddc052e30	*Identifier:* bindto
php:2	c9fd093127e1386a4cd768749d42fe148a87e9e2	2cca43b3abd47835177f4c0562405f1bcfe6c7ac	*Location:* File: main/streams/userspace.c, Line: 284, Type: FunctionDef
php:3	13842eda375a87543650fbf95ffc64f00f3586fc	6df761b7ffc0f54d9d282c394a2a5e53c3d036ef	*Location:* File: main/streams/streams.c, Line: 362, Type: FunctionDef

leading to fewer fitting scenarios (108 from automated analysis, with more than 80 excluded due to outdated libraries, and 3 final scenarios identified through manual analysis).

Overall, we successfully identified 24 different evaluation scenarios across all projects, as shown in Table 1. Each scenario includes specifications for the donor version, the target version, the slicing criterion based on the new functionality, and an associated test case. The slicing criteria were selected based on the descriptions in the commit messages. For example, the commit from scenario **scrcpy:1** from Table 1 adds a functionality for mirroring the clipboard of the smartphone device to the computer and a new accompanying test file (`test_device_event_deserialize.c`). This new functionality is implemented with around 80 new lines of code in a `*.c` and a `*.h` file. Due to the existing C unit test case, we use its ***location*** (the function definition of the `test_deserialize_cliboard_event` function) as slicing criterion. Detailed descriptions of all scenarios can be found in our ***replication package***.[5] These diverse scenarios enable us to comprehensively evaluate the efficacy of our approach in various software transplantation scenarios and externally validate the results, effectively addressing **RQ4**.

4.1.2. Evaluation process

We developed an automated evaluation algorithm (see Algorithm 7) to ensure the replicability of our evaluation. This algorithm requires the following input parameters: the commit hash of the *donorVersion* and *targetVersion* of the program, a slicing criterion SC, and a testCase that validates the correctness of the slice. The algorithm's first step

[5] https://doi.org/10.5281/zenodo.7565855.

involves checking out, installing, and validating the *donorVersion* and the *targetVersion*. We run and document all existing test cases from both versions, including the chosen testCase. For a meaningful evaluation regarding **RQ4**, the selected testCase should succeed in the *donorVersion* and fail in the *targetVersion* (since the latter is missing the expected functionality).

In the subsequent step, we import the *donorVersion* as a vaCPG and validate its accuracy to address **RQ1**. Next, we apply the slicing approach (Fig. 7) using the vaCPG and the defined `SC` as input. To answer **RQ2**, we measure the size of both the input program `P` and the resulting slice `P'`, using the same measurement method (counting the top-level nodes in the vaCPG, as other measurements are not reliable due to the code markup required for the differencing process).

Algorithm 7: Evaluation

```
Data: donorVersion, targetVersion, SC, testCase
Result: results
/* Checkout and install donor and target        */
1 testResultsD = runTests(donorVersion, testCase);
2 testResultsT = runTests(targetVersion, testCase);
3 P = importAsVaCPG(donorVersion);
4 validateVaCPG(P);
5 P' = slice(P , SC);
6 mergedTarget = analysisAndMerge(targetVersion, P');
7 testResultsM = runTests(mergedTarget, testCase);
8 diffs = diff(mergedTarget, donorVersion)
9 return results[testResultsD, testResultsT, testResultsM, diffs];
```

For **RQ3**, we record the duration of each step in Algorithm 7. Once the slicing process is completed, a multi-step analysis and merge of the *targetVersion* and its slice `P'` are triggered. The mergedTarget contains the original source code of the *targetVersion*, along with all new lines from the slice.

The final steps of Algorithm 7 validate the correctness of the results to answer **RQ4** in two different ways. First, the algorithm runs and documents the test cases of the *targetVersion*, as well as the testCase selected to validate the slice. This time, we expect the original test cases to behave the same as in the first run (documented in tResultsT), except for testCase, which should now succeed (as the tested functionality should now be present). In addition, Algorithm 7 creates a diff using the git diff tool (Conservancy, 2023) between the original *targetVersion* and the mergedTarget. We use this diff to manually inspect the changes made by the transplantation process. Finally, the algorithm returns the results of all test runs, as well as the diff.

4.2. Results

We repeated the evaluation process described in Algorithm 7 three times on a virtual machine with Ubuntu 18.0.6 64-bit, 16 GB RAM, and a 4-core processor with 2.4 GHz. Fig. 14 shows a summary of our evaluation results. The left Y-axis represents the process duration in seconds, and the right Y-axis represents the size of the slice in relation to the full program as a percentage. The X-axis contains 24 grouped entries for each scenario (project name followed by a number), where the bars represent the process durations, and the black line represents the slice's relative size. The figure contains three different bar types: The first bar (dark gray) represents the donor's transformation into the vaCPG, the second bar (orange) represents the slicing process, and the third bar (purple) represents the transplantation process of the slice into the *targetVersion*. We intentionally do not use a logarithmic scale, as this would distort the proportions. Instead, we provide a much more fine-grained analysis and additional visualizations in our replication package.

Answer to **RQ1**: Our vaCPG accurately represented the syntactically correct source code of all 24 different program versions, achieving a **100% accuracy rate**. However, we encountered some cases where parts of the code were intentionally designed to be syntactically incorrect or contained errors. For example, in the PHP code, there were two lines consisting of the words "'you"' and "'lose"' to intentionally break the compilation process when a configuration is missing. We included this special case in our custom grammar rules as an additional rule, like the ones described in Section 3.1.2, as it serves a dedicated purpose. However, in another case within the PHP project, a semicolon was missing, resulting in a malformed statement. We did not include this case (resulting in the statement being missing in the vaCPG), as it was a mistake that was later fixed in a subsequent version and did not affect our slicing approach. Additionally, while all four programs could be represented with the current grammar of the vaCPG, each of them required adjustments to the initial grammar due to their distinct coding styles. These adjustments took minutes to hours to complete, depending on the complexity of their coding style.

Answer to **RQ2**: The resulting slice sizes range from 2% to 76% of the donor program, with an average size of about 17% across all scenarios. Notably, the smallest project, Silver Searcher, produced the largest slice, while the largest project, PHP, resulted in slices that take only 11% to 14% of the original program. As expected, slicing criteria of type *location* result in smaller slices than the other types, due to their limited number of entry points. Compared to the average slice size of 33% reported in previous studies (Binkley and Harman, 2004), our resulting slices are notably smaller. This indicates that our slicing approach can significantly reduce the amount of code (**on average by around 83%**) that needs to be considered for subsequent activities.

Answer to **RQ3**: The slicing process took between 20 and 5300 s, with an average duration of about 1000 s over all scenarios. The duration is correlated with the resulting slice's size, where large slices have a longer duration than smaller slices. The required import as vaCPG took between 60 and 410 s, with an average duration of about 130 s. This duration always includes the result's validation, where the vaCPG is transformed back to code and compared with the original input. The import duration is correlated with the project's size, where bigger projects need longer to import. The export of the slice back to code took between 5 to 18 s, with an average duration of about 10 s. In summary, all steps required to produce a syntax-preserving slice in code format **range between minutes and less than two hours**. We argue that this is a practical time frame, especially given that the process is fully automated. However, the numbers also indicate that larger projects will potentially take longer to be imported and sliced. For instance, initial experiments with BusyBox and the Linux kernel confirm this, where both take several hours for the transformation into the vaCPG.

Answer to **RQ4**: Our approach successfully identified and transplanted a functionality in **24 cases over four different C-based projects**. The automated transplantation requires 60 s on average, additionally to the import as vaCPG and the slicing process. While this shows that our novel syntax-preserving slicing approach is usable for software transplantation in a real world context, the transplantation itself is limited to the assumptions described in Section 3.3. Further, the selected input projects are relatively small, so a scalability to large systems still needs to be demonstrated.

4.3. Threats to validity

While our evaluation provides valuable insights into the practical applicability of our syntax-preserving slicing approach for C-based SPLs, there are potential threats to the validity of our findings.

Limited Project Scope: Our evaluation focuses on a specific set of real-world open-source projects that meet our predefined criteria. While these projects were carefully selected to represent various application domains such as file search, smartphone mirroring, and programming languages, the results may not be generalizable to all software systems, particularly large-scale and complex projects. To address this limitation, we conducted additional experiments using

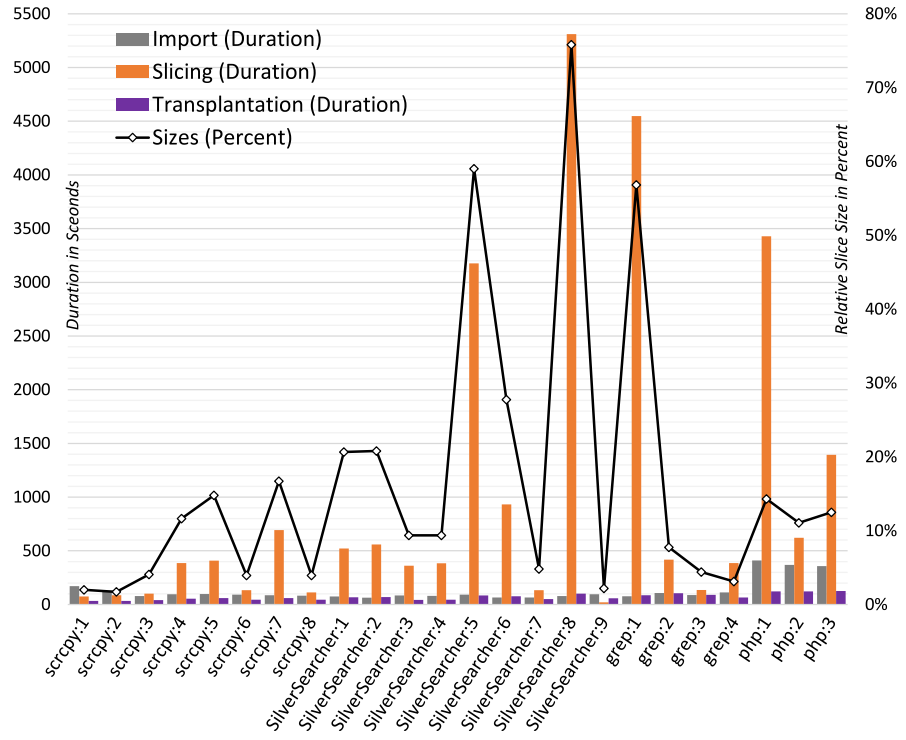

Fig. 14. Relative slice size and duration in seconds for the different processes per scenario.

larger-scale systems like the Linux kernel and BusyBox to assess the applicability of our approach in such contexts. Although we could not validate the correctness of the slices due to missing test cases, these experiments yielded encouraging insights into the scalability of our approach, where the analyses take hours instead of minutes.

Test Case Suitability: The correctness validation of the slice depends on the suitability of the existing test cases from the donor version. In cases where the test cases are insufficient or fail to comprehensively cover the transplanted functionality, the evaluation may not provide an accurate depiction of the slice's behavior in the target version. To mitigate this risk to some extent, we supplemented our evaluation with manual analyses of the test results and the transplantation outcomes. This involved examining commit messages, commit diffs, and conducting code inspections to enhance the validation process.

Tool and Environment Dependencies: The evaluation was conducted using our custom tool and a specific virtual machine environment. Variations in tools, programming environments, or hardware configurations may impact the performance and results of the slicing process. To address this potential concern, we executed our experiments on various machines with diverse hardware specifications to confirm the overall trend of our performance measurements.

Limited Transplantation Scenarios: Our evaluation included a limited number of transplantation scenarios. While the results are promising, further investigations with more diverse projects and use cases are needed to strengthen the generalizability of our approach. We see this as part of our future work, where we plan to improve the capabilities of our transplantation approach, so that the number of fitting scenarios can be increased.

Despite these potential threats, our evaluation demonstrates the feasibility and usefulness of our syntax-preserving slicing and transplantation approach for C-based SPLs. It goes beyond the typical evaluation of slicing approaches, where only time consumption and slice size are measured. Instead, we utilize test cases and manual inspection to also evaluate the content of the calculated slices. Moreover, while our transplantation approach is still a limited prototype, we used real-world software to demonstrate its applicability, providing valuable insights and paving the way for future research and enhancements.

5. Related work

Our approach consists of two main steps: creating a vaCPG and conducting the syntax-preserving slicing algorithm. Due to its preliminary nature and distinctiveness, we exclude a broader discussion of related work for software transplantation (Barr et al., 2015; Petke et al., 2014). Therefore, we structure the related work section into two parts: In the first part, we discuss related approaches constructing a graph-like code representation of C and C preprocessor code. The code representation must contain relevant relations required for the application of slicing, like an AST or a comparable structure. Further, we only discuss approaches explicitly taking C and the C preprocessor into account, as this causes the main challenge for a syntax-preserving slicing approach for C-based SPLs. As a consequence, we do not discuss approaches that create a variability-aware code representation for other languages than C (like SPLLift Bodden et al., 2013, its predecessor by Brabrand et al., 2012 (only CIDE and Java)), and approaches that do not model at least the information of an AST (like srcml Collard et al., 2011). For a more in-depth discussion of why the vaCPG code representation is essential, including concrete code and graph examples and their comparisons, we refer to our previous work (Gerling and Schmid, 2020).

In the second part, we discuss related program slicing approaches for SPLs realized with C and the C preprocessor. Our focus is specifically on approaches explicitly accounting for the challenges posed by the combination of C and the C preprocessor, which represent the primary hurdle for a syntax-preserving slicing approach. As a consequence, we do not discuss approaches conducting program slicing for SPLs implemented with other languages than C (like the approach by Angerer et al., 2019) and approaches analyzing only the preprocessed version of the program (like CSlicer Li et al., 2017, Unravel Lyle and Wallace, 1997, Sprite Bent et al., 2000, and CodeSurfer Bent et al., 2000).

5.1. Code representation

The tool **SuperC** (Gazzillo and Grimm, 2012) generates a variability-aware AST from preprocessor-annotated code. It resolves preprocessor

annotations, except from conditionals, during a lexical analysis step. At this step, it replaces include statements with the included code and macro invocations with their respective expansions. `#ifdef`-statements are represented by static choice nodes inside the AST. These choice nodes contain two child-nodes, one for the true condition and one for the false condition.

While SuperC's code representation allows for some variability-aware static analyses, it is not suitable for syntax-preserving slicing, as it removes the original C preprocessor statements during the lexical analysis step. As a result, the syntax of the original C code is also changed. For a syntax-preserving slicing approach all the preprocessor statements themselves are required as a basis for setting them either as slicing criterion or for retrieving them in the resulting slice. Further, the original syntax of the C code cannot be recovered without additional information, like a change tracing from the original code to the transformed results. Finally, SuperC creates only an AST representation of the code, other relations that are available in our approach are missing. While there is one recent extension of SuperC which allows to conduct additional data-flow analysis (Schubert et al., 2022), this newer approach still inherits the same limitations as SuperC regarding syntax-preservation.

The tool **TypeChef** (Kenner et al., 2010; Kästner et al., 2011) creates a variational AST from preprocessor annotated code, like SuperC does. Also similar to SuperC, TypeChef's AST is built by resolving preprocessor annotations: Include statements are replaced with their included code, macro invocations are replaced with their respective expansions, preprocessor conditionals are represented by choice nodes. The main difference from TypeChef to SuperC are the underlying implementation technologies and slightly differing goals: While SuperC aims at performance and has no specific use case, TypeChef is intended for type checking and computes a fully typed AST. Further, TypeChef also adds presence conditions to each node in the AST, stating their required configuration option if existing.

TypeChef's code inherits the same limitations as SuperC's: It allows for some variability-aware static analyses, but it is not suitable for syntax-preserving slicing, as it removes the original C preprocessor statements during the lexical analysis step. As a result, the syntax of the original C code is also changed. For a syntax-preserving slicing approach all the preprocessor statements themselves are required as a basis for setting them either as slicing criterion or for retrieving them in the resulting slice. Further, the original syntax of the C code cannot be recovered without additional information, like a change tracing from the original code to the transformed results. Finally, TypeChef creates only an AST representation of the code, other relations that are available in our approach are missing. While there are some newer extensions which allow to conduct control- and data-flow analysis (Liebig et al., 2013; Alexander von Rhein et al., 2018), these newer approaches (and all other approaches based on TypeChef's AST) still inherit the same limitations as TypeChef regarding the syntax-preservation.

The tool **CRefactory** (Garrido and Johnson, 2013; Garrido, 2005; Garrido and Johnson, 2003) creates an AST for the purpose of refactoring C and C preprocessor code. The tool implements a dedicated pseudo-preprocessor that analyses macro invocations and conditional compilation directives without removing both of them completely. Instead, CRefactory replaces macro invocations with the macro content, but also adds CppDefine nodes inside the AST. The relations of macro definitions and calls are stored in a table. Conditional compilation statements also have dedicated nodes in the AST. However, both of these nodes do not always represent the original code, as CRefactory only supports preprocessor statements that appear at specific locations, like, before and after a declaration, but not inside of it. As a consequence, CRefactory transforms the original code so that it fits into these rules. The transformations are stored, so that a transformation back to the original code is possible.

In contrast to the previous two approaches, CRefactory preserves some of the original C preprocessor statements, depending on their location. If all preprocessor code within a program followed CRefactory's rules, the resulting program representation would be mostly syntax preserving, except for structural relations from and to preprocessor statements, as they are not existent in the approach. However, as CRefactory's rules are relatively strict, its code representation is potentially not syntax-preserving for real-world projects. Nevertheless, the tool maintains a change tracking for all of its adaptations to the code, so the original syntax can be restored. This makes the approach potentially fitting as code representation for a syntax-preserving slicing approach, although some changes and additions would be needed: An addition of structural relations for C preprocessor statements, an analysis of control- and data-flow relations, and a functionality to map the macro tables with the code statements during the slicing process.

In addition to CRefactory, there exist several other approaches targeting **refactoring of C and C preprocessor** code. However, CRefactory is the only approach that allows a complete representation of the original preprocessor code in the AST, or alternatively stores the transformations necessary to restore the original code. In the following, we therefore discuss these other approaches with less detail on the single approach but with regard to a categorization proposed by Liebig et al. (2017). In their paper, they empirically analyzed state of the art refactoring approaches and categorized them based onto their handling of variability into single-variant, variant-based, limited-pattern-based, and heuristics-based approaches. Most of the listed approaches belong to the single-variant category, which completely ignores preprocessor code, and are therefore out of scope for our related work discussion. Furthermore, the approach proposed by Liebig et al. will also not be discussed, as it uses the code representation provided by TypeChef, which we already discussed in the beginning of the section.

Variant-based refactoring approaches like **XRefactory** (Vittek, 2003), **CScout** (Spinellis, 2010), and **Proteus** (Waddington and Yao, 2007) produce different ASTs for all different program variants potentially affected by a refactoring operation. The configurations must be given manually by a developer and the resulting ASTs do not contain preprocessor if statements or any kind of variability relations. A dedicated storage of presence conditions, like CRefactory maintains, is also not existent in those approaches. As a result, the code representations of variant-based refactoring approaches are not suitable as input for syntax-preserving slicing.

Limited-pattern-based refactoring approaches like **DMS** (Baxter, 2002) build an AST containing preprocessor elements, but only when the directives follow a limited number of specific patterns. This is similar to the approach of CRefactory and therefore inherits the same limitations. However, limited-pattern-based approaches are not able to automatically transform the source code so that it fits into the pattern, like CRefactory does. As a consequence, the representation of preprocessor elements is strictly limited to those that follow the patterns, deviating elements must be transformed manually. Furthermore, a representation of control, data, and variability relations still needs to be implemented.

Heuristics-based refactoring approaches like **CRefactory** and **Coccinelle(Yacfe)** (Padioleau et al., 2008; Padioleau, 2009), also rely on restricting the preprocessor code to specific patterns. However, they utilize additional heuristics to handle those statements that do not adhere to the patterns. While the CRefactory approach explicitly tracks changes to the source code and has a detailed documentation of the used heuristics and their effects, the documentation of the Yacfe parser is rather sparse. The tool employs a complex combination of customized grammar rules with so called fresh tokens, that represent dedicated rules for handling different preprocessor patterns, different views for determining the current context of a token, heuristics based on identifiers and indentation, and the specification of a configuration file with manually defined macro definitions. Based on their evaluation (Padioleau, 2009), the approach requires on average 56 manual hints per program, while still parsing on average 4% of the source code wrong. This makes the approach not suitable for automated syntax-preserving program slicing.

While heuristics-based approaches for building ASTs are not inappropriate in general as input for syntax-preserving program slicing, it is crucial that the used heuristics are transparent and eventual changes to the code can be reversed. Further, as we aim at a fully automated slicing approach, suitable code representations should not require extensive manual specification of input configurations or manual adaptations to the source code.

5.2. Slicing approaches

Kanning and Schulze (2014) introduced a concept and a prototype implementation for a variability-aware program slicer. This slicer is specifically designed for C software incorporating variability through conditional compilation. Their approach hinges on a variability-aware program dependence graph, which includes presence conditions associated with the edges. To generate this graph, they employ TypeChef (Kenner et al., 2010; Kästner et al., 2011), which we discussed in Section 5.1. Kanning and Schulze's approach follows a forward slicing methodology, similar to traditional forward slicing methods. It aims to analyze statements influenced by variables at specified locations, considering both control and data flows. However, their approach also determines the configurations under which a particular statement is affected.

Kanning and Schulze's approach stands out as one of the relatively few methods dedicated to variability-aware program slicing in C software systems. Unlike our approach, they have implemented a forward slicing methodology, which, while valuable for certain purposes, does not align with our use case of software transplantation. This forward slicing approach primarily focuses on analyzing the effect of a statement but not its dependencies. Nonetheless, the challenges encountered in both approaches are comparable. Notably, Kanning and Schulze's approach does not prioritize preserving the syntax of the original program. Instead, it relies on TypeChef to resolve preprocessor code. Consequently, the computed slices cannot encompass preprocessor statements, such as macro definitions or include statements. Moreover, setting preprocessor statements or configuration options as slicing criteria is not supported. Additionally, due to its prototypical nature, this approach is incapable of calculating slices extending beyond a single function, despite the graph generated by TypeChef having the potential to parse an entire program. Consequently, the real-world applicability of Kanning and Schulze's approach is significantly restricted, in contrast to our approach. Lastly, their approach is confined to a reduced subset of C language constructs. For instance, it lacks the capability to handle global variables, pointers, parameters, or externally defined functions and function-like macros, all of which are encompassed in our analysis.

The **srcSlice** approach, as introduced by Alomari et al. (2014), employs a forward decomposition slicing technique tailored for C software. Much like the previously discussed approach, srcSlice identifies statements influenced by a chosen variable. However, srcSlice also employs a form of decomposition slicing where no specific slicing criterion is provided. Instead, the approach computes slices for all existing variables within a given file. Notably, srcSlice's methodology for calculating these slices differs from the typical approach of traditional slicing methods. It does not rely on the analysis of an existing program dependence graph or similar input. Instead, srcSlice leverages the output generated by the srcML tool (Collard et al., 2011), which employs a custom markup language to model source code. This model does not encompass control or data-flow relations but categorizes source statements based on their syntax. srcSlice builds upon this model and calculates data and control dependencies locally for each encountered variable.

While srcML's code representation is fully syntax-preserving, encompassing a representation of preprocessor code, srcSlice does not utilize this information. Consequently, despite sharing a comparable starting point to our approach, srcSlice lacks the same capabilities as our approach concerning the handling of preprocessor code or comments. It does not consider the preprocessor's influence when computing dependencies, and the resulting slice cannot include preprocessor code or comments. Furthermore, srcSlice serves a different use case. It computes a forward slice and employs decomposition slicing, both of which are ill-suited for a software transplantation scenario. Additionally, the implementation of decomposition slicing removes control over the slicing outcome, unlike traditional slicing approaches and even more so compared to our approach, as it does not permit a custom slicing criterion to be provided. Lastly, the desired quality attributes of srcSlice's results contrast with ours. The authors emphasize srcSlice's speed as one of its primary advantages, prioritizing speed over completeness. Consequently, the tool can swiftly slice even large-scale systems, but the results may omit crucial components. In contrast, our approach is slower but prioritizes comprehensive and transparent results.

The approach presented by Vidács et al. (2009) combines the commercial program slicer previously known as CodeSurfer (GrammaTech, 2023) designed for C software with an extension tailored for C preprocessor macro slicing. They implement two distinct slicing approaches: one for forward slicing and another for backward slicing. Both of these approaches follow distinct methodologies and employ different slicing criteria. However, both approaches rely on a separately computed slice of the C portion of the software, in conjunction with a slice exclusively derived from macro dependencies. The C software slice is computed using CodeSurfer and adheres to a traditional slicing methodology that utilizes a system dependence graph constructed from preprocessed code. In contrast, the macro slicer employs a macro dependency graph generated through the Columbus framework (Vidacs et al., 2004), which models all macro definitions and their invocations within the source code.

For the calculation of a forward slice, only a macro definition can be set as the slicing criterion. The approach then calculates all other macro definitions that can be affected by this macro. The resulting set of macro definitions is then matched one-by-one with each node in the C system dependence graph that belongs to a function, by analyzing their position and the contained characters. If a match is found, CodeSurfer is invoked to calculate a classical forward slice for the C node given the matched node as the slicing criterion. The final result is a union of all found macro definitions and all calculated C slices.

To calculate a backward slice, a C language element must be set as the slicing criterion in the form of a traditional tuple consisting of location and variable. CodeSurfer then computes a traditional backward slice for the preprocessed C code. This slice is matched with the macro dependency graph in a manner similar to the forward slicing approach: Each C node within a function is matched with each macro definition in the macro dependency graph. They are matched based on their location and character content. If they match, allowing the macro to affect the node's content, the macro definition is included in the slicing results. Furthermore, the dependencies of the macro are analyzed to add all other macro definitions to the slice that can influence the content of this macro. The final slicing result is a union of the C slice and the identified macro definitions.

Vid'acs et al.'s approach bears the closest resemblance to ours, as they also aim for a combined slicing approach for C and C preprocessor code. Furthermore, they have implemented a backward slicing approach. However, unlike our approach, both languages are kept in separate models, and the slicing process is also conducted separately for each language model. Additionally, the model for the preprocessor code contains only macro definitions and no other preprocessor elements, such as ifdef or include statements. Consequently, their approach is not a slicer for C and C preprocessor like our approach, but rather an enhancement of an existing C slicer to include relations to preprocessor macros exclusively. Therefore, their slices may miss relevant parts of the program, which can be contained in our slicing results, like include or ifdef statements. Furthermore, as the existing C slicer relies on a

preprocessed version of the code, the resulting C slice lacks code for all non-selected configurations. Finally, the macro slicing extension is only able to identify relations within C functions, excluding all other statements, such as global variable declarations or type definitions. This results in even more potentially relevant statements missing from the calculated slice in comparison to our approach.

The tool Ghinshu (Livadas and Small, 1994) is a graphical software maintenance environment that explicitly targets an integrated slicing approach for both C and C preprocessor code. It constructs a system dependence graph from the source code using a parser to calculate, among other things, backward slices. Ghinshu employs a special preprocessor named Ghinshu preprocessor to keep track of macro definitions, their invocations, and their expansions in the source code, which is comparable to the macro dependency graph used in Vid'acs et al.'s approach (Vidács et al., 2009). However, Ghinshu does not maintain two separate graphs but instead incorporates custom nodes with macro information into the affected nodes of the system dependence graph. To calculate a backward slice, Ghinshu requires a slicing criterion in the form of a location and a C variable. It then utilizes the traditional methodology to compute the slice. The resulting C code lines of the slice are graphically highlighted, and connections to the related preprocessor define statements are displayed.

The motivation behind incorporating backward slicing with macro definitions in Ghinshu closely aligns with our motivation, given that the lack of preprocessor analysis was a longstanding issue even three decades ago. Ghinshu aimed to address this concern, with macro analysis being its initial step. While the general approach, involving lexical analysis, parsing, AST creation, and the calculation of control and data dependencies to construct a system dependence graph, is conventional and akin to our approach, Ghinshu introduced a distinct concept by utilizing a specialized preprocessor. However, the project was not further developed, resulting in the absence of other preprocessor elements such as include or conditional compilation statements in the code representation, slicing criteria, or slicing results, unlike our approach. Furthermore, some descriptions of Ghinshu suggest that the system dependence graph was constructed for a subset of the C language, excluding certain elements like typedefs or function pointers, and that it may not handle source code spanning multiple files (Livadas and Rosenstein, 1994). This represents a significant reduction in capabilities compared to our approach, which models the entire C language and is not restricted to a single source file.

6. Conclusion

We have introduced a novel syntax-preserving program slicing approach for C-based SPLs based on the vaCPG, a joint code representation of C and CPP code. Our approach stands out as the first to enable the computation of program slices through an integrated analysis of both C and CPP code, without any changes to the original program syntax. Consequently, the resulting program slices are true subsets of the SPL, making them suitable inputs for variability-aware analyses and other dedicated applications.

6.1. Summary

Through an empirical evaluation using four different real-world projects of varying sizes, we demonstrated the **effectiveness** of our syntax preserving slicing and transplantation approach in **24 distinct scenarios** (**RQ4**). Our novel data structure, the vaCPG, was able to model the syntactically correct C and CPP source **100% accurately** in each of these scenarios (**RQ1**). Remarkably, our approach successfully completed the slicing and transplantation in a reasonable time frame, taking between **minutes** and **less than two hours** for completion (**RQ3**). The resulting slices, on average, accounted for only **17% of the original program's size**, showcasing a significant reduction in complexity when utilized for dedicated analyses (**RQ2**).

Additionally, our work goes beyond syntax-preserving slicing and explores the practical application of software transplantation. While our preliminary implementation without the reliance on test cases marks just the first steps in this direction, we view it as a promising avenue for future research. In conclusion, our syntax-preserving slicing approach, combined with the vaCPG representation, holds the potential to unlock new possibilities in C-based SPL analysis and software transplantation, offering insights into the behavior of SPLs and facilitating the reusability of functionality across versions.

6.2. Future work

While our current approach is language dependent and limited to C and CPP code, we still see a lot of potential to address this limitation in the future. For example, the vaCPG could be extended to also contain information from the build system or documentation artifacts, as long as they follow a context-free grammar. When the mapping of these artifacts to the source code fragments is also parseable, the graph could be extended with new relations, like DOCUMENTS or BUILDS. Consequently, these could be explored with new rules during the slicing process, and then transplanted together with the source code.

Moreover, our approach does not take the impact of configuration options into account when transplanting configurable software. This could result in conflicts (Nadi et al., 2015), mismatches (El-Sharkawy et al., 2017), or dead variable code (Tartler et al., 2011) (variable code that is never included in a configured variant). Here, we also see a very interesting use case in integrating existing configuration or variability models into the vaCPG, to model the relations of the different existing configuration options and their impact over different artifact types.

Finally, our work can be valuable in the broader context of SPL analysis: The vaCPG offers interesting possibilities for analyzing the relationships within the source code, such as identifying sections of code with similar structure and content. This could be beneficial for variability mining (Kästner et al., 2014) and clone detection (Komondoor and Horwitz, 2001; Roy and Cordy, 2007), to foster adopting a more systematic SPL engineering approach.

CRediT authorship contribution statement

Lea Gerling: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A complete replication package is available at https://doi.org/10.5281/zenodo.7565855. It contains the virtual machine used for the evaluation, the collected data with additional analyses and visualizations, and the source code of our tool. The approach is also available as open source tool at https://github.com/LPhD/Jess.

References

Aho, A.V., Sethi, R., Ullman, J.D., 2007. Compilers: Principles, Techniques, & Tools, second ed. Pearson/Addison Wesley, Boston.

Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, Sven Apel, 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. ACM Trans. Softw. Eng. Methodol. (TOSEM) 27 (4), 18:1–18:33.

Alomari, H.W., Collard, M.L., Maletic, J.I., Alhindawi, N., Meqdadi, O., 2014. srcSlice: Very efficient and scalable forward static slicing. J. Softw.: Evol. Process 26 (11), 931–961. http://dx.doi.org/10.1002/smr.1651.

Angerer, F., Grimmer, A., Prähofer, H., Grünbacher, P., 2019. Change impact analysis for maintenance and evolution of variable software systems. Autom. Softw. Eng. 26, 1–45. http://dx.doi.org/10.1007/s10515-019-00253-7.

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C., 2011. Semistructured merge: rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering - SIGSOFT/FSE '11. ACM Press, Szeged, Hungary, p. 190. http://dx.doi.org/10.1145/2025113.2025141, URL http://dl.acm.org/citation.cfm?doid=2025113.2025141.

Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J., 2015. Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, Baltimore, MD, USA, pp. 257–269.

Baxter, I.D., 2002. DMS: Program transformations for practical scalable software evolution. In: Proceedings of the International Workshop on Principles of Software Evolution. pp. 48–51.

Bent, L., Atkinson, D.C., Griswold, W.G., 2000. A Qualitative Study of Two Whole-Program Slicers for C. Technical Report.

Bergey, J., Clements, P., Cohen, S., Donohoe, P., Jones, L., 1998. DoD Product Line Practice Workshop Report:. Tech. rep., Defense Technical Information Center, Fort Belvoir, VA, http://dx.doi.org/10.21236/ADA346252, URL http://www.dtic.mil/docs/citations/ADA346252.

Binkley, D.W., Harman, M., 2004. A survey of empirical results on program slicing. Adv. Comput. 62, 105–178.

Bitzer, J., Schröder, P.J., 2006. The Impact of Entry and Competition by Open Source Software on Innovation Activity. In: The Economics of Open Source Software Development. Elsevier, pp. 219–246. http://dx.doi.org/10.1016/B978-044452769-1/50011-1.

Bodden, E., Toledo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M., 2013. SPLLIFT — Statically analyzing software product lines in minutes instead of years. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 355–364.

Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P., 2012. Intraprocedural dataflow analysis for software product lines. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development - AOSD '12. ACM Press, Potsdam, Germany, pp. 13–24. http://dx.doi.org/10.1145/2162049.2162052.

Buffenbarger, J., 1995. Syntactic software merging. In: Goos, G., Hartmanis, J., Leeuwen, J., Estublier, J. (Eds.), Software Configuration Management. In: Series Title: Lecture Notes in Computer Science, vol. 1005, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 153–172. http://dx.doi.org/10.1007/3-540-60578-9_14, URL http://link.springer.com/10.1007/3-540-60578-9_14.

Cavalcanti, G., Borba, P., Seibt, G., Apel, S., 2019. The impact of structure on software merging: semistructured versus structured merge. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1002–1013.

Collard, M.L., Decker, M.J., Maletic, J.I., 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. IEEE, Williamsburg, VA, USA, pp. 173–184. http://dx.doi.org/10.1109/SCAM.2011.19.

Conservancy, S.F., 2023. Git Diff. URL https://git-scm.com/docs/git-diff.

Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K., 2013. An exploratory study of cloning in industrial software product lines. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, Genova, pp. 25–34. http://dx.doi.org/10.1109/CSMR.2013.13.

El-Sharkawy, S., Krafczyk, A., Schmid, K., 2017. An empirical study of configuration mismatches in linux. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A. ACM, Sevilla Spain, pp. 19–28. http://dx.doi.org/10.1145/3106195.3106208, URL https://dl.acm.org/doi/10.1145/3106195.3106208.

Erik Andersen, T.S.F.C., 2023. BusyBox Homepage. URL https://www.busybox.net/.

Ernst, M., Badros, G., Notkin, D., 2002. An empirical analysis of c preprocessor use. IEEE Trans. Softw. Eng. 28 (12), 1146–1170. http://dx.doi.org/10.1109/TSE.2002.1158288, URL http://ieeexplore.ieee.org/document/1158288/.

Ernst, N.A., Easterbrook, S., Mylopoulos, J., 2010. Code forking in open-source software: a requirements perspective. arXiv:1004.2889 [cs].

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9 (3), 319–349. http://dx.doi.org/10.1145/24039.24041, URL https://dl.acm.org/doi/10.1145/24039.24041.

Foundation, F.S., 2023. Grep Repository. URL https://github.com/neverware-mirrors/grep.

Free Software Foundation, 2023. The GNU C preprocessor. URL https://gcc.gnu.org/onlinedocs/cpp/.

Garousi, V., Felderer, M., Karapıçak, Ç.M., Yılmaz, U., 2018. Testing embedded software: A survey of the literature. Inf. Softw. Technol. 104, 14–45. http://dx.doi.org/10.1016/j.infsof.2018.06.016, URL https://linkinghub.elsevier.com/retrieve/pii/S0950584918301265.

Garrido, A., 2005. Program refactoring in the presence of preprocessor directives (Ph.D. thesis). University of Illinois, http://dx.doi.org/10.35537/10915/4167, URL http://sedici.unlp.edu.ar/handle/10915/4167.

Garrido, A., Johnson, R., 2003. Refactoring C with conditional compilation. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.. IEEE Comput. Soc, Montreal, Que., Canada, pp. 323–326. http://dx.doi.org/10.1109/ASE.2003.1240330, URL http://ieeexplore.ieee.org/document/1240330/.

Garrido, A., Johnson, R., 2013. Embracing the c preprocessor during refactoring: Refactoring C and the C preprocessor. J. Softw.: Evol. Process 25 (12), 1285–1304. http://dx.doi.org/10.1002/smr.1603, URL https://onlinelibrary.wiley.com/doi/10.1002/smr.1603.

Gazzillo, P., Grimm, R., 2012. SuperC: Parsing all of c by taming the preprocessor. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 323–334.

Gerling, L., Schmid, K., 2019. Variability-aware semantic slicing using code property graphs. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A. ACM, Paris France, pp. 65–71. http://dx.doi.org/10.1145/3336294.3336312.

Gerling, L., Schmid, K., 2020. Syntax-preserving slicing of c-based software product lines: an experience report. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 1–5.

GitHub, 2023. GitHub. URL https://github.com.

GrammaTech, 2023. CodeSonar. URL https://www.grammatech.com/codesonar-cc.

Greer, G., 2023. Silver Searcher Repository. URL https://github.com/ggreer/the_silver_searcher.

Group, T.P., 2023. PHP Interpreter Repository. URL https://github.com/php/php-src.

Harman, M., Langdon, W.B., Weimer, W., 2013. Genetic programming for reverse engineering. In: 2013 20th Working Conference on Reverse Engineering. WCRE, IEEE, Koblenz, Germany, pp. 1–10. http://dx.doi.org/10.1109/WCRE.2013.6671274, URL http://ieeexplore.ieee.org/document/6671274/.

Horwitz, S., Prins, J., Reps, T., 1989. Integrating noninterfering versions of programs. ACM Trans. Program. Lang. Syst. 11 (3), 345–387. http://dx.doi.org/10.1145/65979.65980.

Horwitz, S., Reps, T., Binkley, D., 1988. Interprocedural Slicing Using Dependence Graphs. Technical Report 756, University of Wisconsin-Madison, 1210 W. Dayton St., Madison, WI 53706.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., Apel, S., 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empir. Softw. Eng. 21 (2), 449–482. http://dx.doi.org/10.1007/s10664-015-9360-1, URL http://link.springer.com/10.1007/s10664-015-9360-1.

Kanning, F., Schulze, S., 2014. Program slicing in the presence of preprocessor variability. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, Victoria, BC, Canada, pp. 501–505. http://dx.doi.org/10.1109/ICSME.2014.82.

Kästner, C., Dreiling, A., Ostermann, K., 2014. Variability mining: Consistent semi-automatic detection of product-line features. IEEE Trans. Softw. Eng. 40 (1), 67–82. http://dx.doi.org/10.1109/TSE.2013.45, URL http://ieeexplore.ieee.org/document/6613490/.

Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T., 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM, Portland, Oregon, USA, pp. 805–824.

Kenner, A., Kästner, C., Haase, S., Leich, T., 2010. TypeChef: toward type checking #ifdef variability in C. In: Proceedings of the 2nd International Workshop on Feature-Oriented Software Development - FOSD '10. ACM Press, Eindhoven, The Netherlands, pp. 25–32. http://dx.doi.org/10.1145/1868688.1868693.

Komondoor, R., Horwitz, S., 2001. Using slicing to identify duplication in source code. In: Goos, G., Hartmanis, J., van Leeuwen, J., Cousot, P. (Eds.), Static Analysis. Vol. 2126, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 40–56. http://dx.doi.org/10.1007/3-540-47764-0_3, URL http://link.springer.com/10.1007/3-540-47764-0_3.

Li, Y., Zhu, C., Rubin, J., Chechik, M., 2017. Semantic slicing of software version histories. IEEE Trans. Softw. Eng. 2 (44), 182–201.

Liebig, J., Apel, S., Janker, A., Garbe, F., Oster, S., 2017. Handling static configurability in refactoring engines. Computer 50 (7), 44–53. http://dx.doi.org/10.1109/MC.2017.212, URL http://ieeexplore.ieee.org/document/7971858/.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M., 2010. An analysis of the variability in forty preprocessor-based software product lines. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on. Vol. 1, IEEE, pp. 105–114.

Liebig, J., Kästner, C., Apel, S., 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development - AOSD '11. ACM Press, Porto de Galinhas, Brazil, p. 191. http://dx.doi.org/10.1145/1960275.1960299, URL http://portal.acm.org/citation.cfm?doid=1960275.1960299.

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C., 2013. Scalable analysis of variable software. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013. ACM Press, Saint Petersburg, Russia, pp. 81–91. http://dx.doi.org/10.1145/2491411.2491437.

Linden, F.v.d., Schmid, K., Rommes, E., 2007. Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering. Springer, Berlin ; New York.

Livadas, P.E., Rosenstein, A., 1994. Slicing in the Presence of Pointer Variables. Technical Report SERC-TR-74-F, Computer Science and Information Services Department, Technical Report SERC-TR-74-F, Computer Science and Information Services Department.

Livadas, P., Small, D., 1994. Understanding code containing preprocessor constructs. In: Proceedings 1994 IEEE 3rd Workshop on Program Comprehension- WPC '94. IEEE Comput. Soc. Press, Washington, DC, USA, pp. 89–97. http://dx.doi.org/10.1109/WPC.1994.341255, URL http://ieeexplore.ieee.org/document/341255/.

LPhD, 2023. CommitAnalysis. URL https://github.com/LPhD/CommitAnalysis/.

Lyle, J.R., Wallace, D.R., 1997. Using the unravel program slicing tool to evaluate high integrity software. Technology 301.

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., Gheyi, R., 2015. The Love/hate relationship with the c preprocessor: An interview study. In: Leibniz International Proceedings in Informatics (LIPIcs). 37, Schloss Dagstuhl – Leibniz-Zentrum f{\"u}r Informatik, Dagstuhl, Germany, p. 24. http://dx.doi.org/10.4230/LIPICS.ECOOP.2015.495, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2015.495. Artwork Size: 24 pages, 779569 bytes ISBN: 9783939897866 Medium: application/pdf Publisher: [object Object].

Mens, T., 2002. A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. 28 (5), 449–462. http://dx.doi.org/10.1109/TSE.2002.1000449, URL http://ieeexplore.ieee.org/document/1000449/.

Nadi, S., Berger, T., Kastner, C., Czarnecki, K., 2015. Where do configuration constraints stem from? An extraction approach and an empirical study. IEEE Trans. Softw. Eng. 41 (8), 820–841. http://dx.doi.org/10.1109/TSE.2015.2415793, URL http://ieeexplore.ieee.org/document/7065312/.

Organization, T.L.K., 2023. The linux kernel archives. URL https://www.kernel.org/.

Padioleau, Y., 2009. Parsing C/C++ Code without Pre-processing. In: de Moor, O., Schwartzbach, M.I. (Eds.), Compiler Construction. In: Series Title: Lecture Notes in Computer Science, vol. 5501, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 109–125. http://dx.doi.org/10.1007/978-3-642-00722-4_9, URL http://link.springer.com/10.1007/978-3-642-00722-4_9.

Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G., 2008. Documenting and automating collateral evolutions in linux device drivers. ACM Sigops Oper. Syst. Rev. 42 (4), 247–260.

Petke, J., Harman, M., Langdon, W.B., Weimer, W., 2014. Using genetic improvement and code transplants to specialise a c++ program to a problem class. In: Genetic Programming: 17th European Conference, EuroGP 2014, Granada, Spain, April 23-25, 2014, Revised Selected Papers 17. Springer, pp. 137–149.

Reps, T., Turnidge, T., 1995. Program Specialization via Program Slicing. Technical Report 1296, University of Wisconsin-Madison.

Reps, T., Yang, W., 1988. The Semantics of Program Slicing. Technical Report 777, University of Wisconsin-Madison.

Rosiak, K., Urbaniak, O., Schlie, A., Seidl, C., Schaefer, I., 2019. Analyzing variability in 25 years of industrial legacy software: an experience report. In: Proceedings of the 23rd International Systems and Software Product Line Conference Volume B - SPLC '19. ACM Press, Paris, France, pp. 1–8. http://dx.doi.org/10.1145/3307630.3342410, URL http://dl.acm.org/citation.cfm?doid=3307630.3342410.

Roy, C.K., Cordy, J.R., 2007. A Survey on Software Clone Detection Research. Technical Report 2007–541, School of Computing Queen's University at Kingston, Ontario, Canada, p. 115.

Schubert, P.D., Gazzillo, P., Patterson, Z., Braha, J., Schiebel, F., Hermann, B., Wei, S., Bodden, E., 2022. Static data-flow analysis for software product lines in C: Revoking the preprocessor's special role. Autom. Softw. Eng. 29 (1), 35, Publisher: Springer.

Seibt, G., Heck, F., Cavalcanti, G., Borba, P., Apel, S., 2022. Leveraging structure in software merge: An empirical study. IEEE Trans. Softw. Eng. 48 (11), 4590–4610. http://dx.doi.org/10.1109/TSE.2021.3123143, URL https://ieeexplore.ieee.org/document/9591645/.

Silva, J., 2012. A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44 (3), 1–41. http://dx.doi.org/10.1145/2187671.2187674.

Spinellis, D., 2010. CScout: A refactoring browser for C. Sci. Comput. Program. 75 (4), 216–231. http://dx.doi.org/10.1016/j.scico.2009.09.003, URL https://linkinghub.elsevier.com/retrieve/pii/S016764230900121X.

Stănciulescu, Ş., Schulze, S., Wąsowski, A., 2015. Forked and integrated variants in an open-source firmware project. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, pp. 151–160.

Sung, C., Lahiri, S.K., Kaufman, M., Choudhury, P., Wang, C., 2020. Towards understanding and fixing upstream merge induced conflicts in divergent forks: An industrial case study. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. Association for Computing Machinery, Seoul, South Korea, p. 10. http://dx.doi.org/10.1145/3377813.3381362.

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., Lohmann, D., 2014. Static analysis of variability in system software: The 90, 000# ifdefs issue. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA, pp. 421–432.

Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W., 2011. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In: Proceedings of the Sixth Conference on Computer Systems - EuroSys '11. ACM Press, Salzburg, Austria, p. 47. http://dx.doi.org/10.1145/1966445.1966451, URL http://portal.acm.org/citation.cfm?doid=1966445.1966451.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. 47 (1), 1–45. http://dx.doi.org/10.1145/2580950.

Vidacs, L., Beszedes, A., Ferenc, R., 2004. Columbus schema for C/C++ preprocessing. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.. IEEE, Tampere, Finland, pp. 75–84. http://dx.doi.org/10.1109/CSMR.2004.1281408, URL http://ieeexplore.ieee.org/document/1281408/.

Vidács, L., Beszédes, Á., Gyimóthy, T., 2009. Combining preprocessor slicing with C/C++ language slicing. Sci. Comput. Program. 74 (7), 399–413, Publisher: Elsevier.

Vimont, R., 2023. Scrcpy Repository. URL https://github.com/Genymobile/scrcpy.

Vittek, M., 2003. Refactoring browser with preprocessor. In: Seventh European Conference OnSoftware Maintenance and Reengineering, 2003. Proceedings.. IEEE Comput. Soc, Benevento, Italy, pp. 101–110. http://dx.doi.org/10.1109/CSMR.2003.1192417, URL http://ieeexplore.ieee.org/document/1192417/.

Waddington, D., Yao, B., 2007. High-fidelity C/C++ code transformation. Sci. Comput. Program. 68 (2), 64–78. http://dx.doi.org/10.1016/j.scico.2006.04.010, URL https://linkinghub.elsevier.com/retrieve/pii/S0167642307000718.

Weiser, M., 1981. Program Slicing. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 439–449.

Yamaguchi, F., 2015. Pattern-Based Vulnerability Discovery (Ph.D. thesis). Göttingen.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, San Jose, CA, pp. 590–604. http://dx.doi.org/10.1109/SP.2014.44.

Zhou, S., Stănciulescu, Ş., Leßenich, O., Xiong, Y., Wąsowski, A., Kästner, C., 2018. Identifying features in forks. In: Proceedings of the 40th International Conference on Software Engineering. ACM, Gothenburg Sweden, pp. 105–116. http://dx.doi.org/10.1145/3180155.3180205.

Lea Gerling is a software engineering researcher specializing in highly configurable C software systems, static analysis (especially program slicing), and software evolution. Her research focuses on software product lines and variability, with specific interests in feature migration, incremental verification, and adaptive software systems.