Program Slicing in the Presence of Preprocessor Variability

Frederik Kanning
TU Braunschweig
Braunschweig, Germany
Email: f.kanning@tu-braunschweig.de

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
Email: sanschul@tu-braunschweig.de

Abstract—Program slicing is a common means to support developers in examining the source code with respect to debugging, program comprehension, or regression testing. While a vast amount of techniques exist, they are mostly tailored to single software systems. However, with the increasing importance of variable and highly-configurable systems, such as the Linux kernel, the number of software variants, subject to analysis, increase dramatically. Consequently, it is infeasible to apply slicing on each variant in isolation. To overcome this problem, we propose variability-aware slicing, a technique that can deal with source code variability, specifically conditional compilation as introduced by the C preprocessor. Particularly, we provide details of our variability-aware dependence analysis for program slicing, point out benefits of our slicing technique, and mention current limitations and future work.

I. INTRODUCTION

Program slicing is a common and well-established method for detailed program analysis, which has been proposed more than three decades ago by Weiser [1], [2]. Basically, program slicing allows to decompose a program with respect to its computational dependencies. As a result, it is possible to inspect the program in more detail, for instance, to assess the (potential) influence of a set of statements (the program slice) on a particular point in the program (the slicing criterion). Meanwhile, numerous techniques have been proposed that extend the original (static) slicing technique of Weiser and thus, support developers in one or more tasks, such as debugging, program comprehension, or regression testing [3], [4].

While existing techniques are very mature and applicable even at large scale for single (monolithic) programs, they are only of limited use for variable software systems. Such systems usually constitute a whole family of related programs that can be derived from a common code base [5]. To this end, configuration options, or features allow users to specify variable parts in such systems. A common means to express this variability are preprocessor directives, in particular conditional compilation, as provided by the C preprocessor tool CPP [6], [7]. While this allows users to tailor a program according to specific needs, it also gives rise to an exponential number of configurations. For instance, the Linux kernel consists of more than 11,000 features, which allow for billions of possible configurations to be compiled and generated on demand.

In this paper we bridge the gap between slicing and source code variability by proposing a technique that supports exploration and static analysis of variable software systems.

Research problem: Obviously, statically analyzing billions of configurations separately is infeasible. On the other hand, current slicing techniques mainly work on preprocessed programs (i.e., a particular configuration), that is, the CPP directives are evaluated and removed before analysis, and thus, variability is not supported. In other words, such kind of analysis is not variability-aware [8]. As a result, these slicing techniques are neither applicable to variable software systems nor can detect effects of variability, such as errors caused by feature interactions [9].

Contribution: The main contribution of this paper is a concept and prototypical implementation for variability-aware intraprocedural slicing that works on annotated C programs (i.e., before preprocessing CPP directives). To this end, we extend and instrument TYPECHEF, a research infrastructure for variability-aware analysis [10], [11]. Particularly, we provide details on our variability-aware dependence analysis and how we use this analysis to compute precise slices that include variability information. Moreover, we motivate our technique by means of an example and possible applications.

While the current implementation exhibits still some limitations, we argue that it provides an important foundation for taming variability and thus, provide efficient and scalable program analysis for variable software systems.

II. BACKGROUND

In the following, we provide information about source code variability using CPP directives. Moreover, we lay the foundations of our slicing technique by introducing the basic concepts of program slicing.

A. Variable Systems with the CPP

The C preprocessor CPP is a tool that is tightly integrated with the C programming language and that allows for developing variable software systems [7]. To this end, the CPP provides capabilities to annotate variable code fragments at arbitrary granularity using conditional compilation (a.k.a. #ifdefs). As an example, we show a program excerpt in Figure 1 that contains two variable code fragments: One variable fragment (Line 6–8) redeclares variable c in the inner scope, while the other one (Line 13–15) assigns a constant value to b. Both fragments are optional and their in-/exclusion is controlled by configuration options (a.k.a. features), by assigning (boolean) values, so-called presence conditions, to each of them. In our
example, we have two optional features FOO and BAR. Moreover, such features can be combined using boolean operators.

B. Program Slicing and PDGs

The general purpose of slicing is to obtain a subset of a program, such that this subset contains all program points (usually statements and expressions) that impact the computations specified by a given slicing criterion [1]. Particularly, this is referred to as backward slicing. In contrast, computing all program points that depend on a criterion is called forward slicing [12]. Particular slicing techniques focus on tracing dependences among program elements. To this end, they exploit a specific data structure called program dependence graph (PDG) [13], [14].

A PDG is a directed multigraph with nodes representing program points and edges representing computational dependences between those points. More specifically, a PDG contains an edge \((P, P')\) if and only if \(P'\) depends on \(P\). Basically, a PDG specifies two kinds of dependences: First, \(P'\) is data-dependent on \(P\), if \(P'\) consumes data defined by \(P\). Second, \(P'\) is control-dependent on \(P\), if the value of \(P\) (which is usually an expression) determines whether \(P'\) is executed. A slicing criterion is a subset of PDG nodes and a (backward) slice is the set of nodes that reach a node of this criterion on some path. Accordingly, a forward slice is the set of all nodes reachable from the criterion. For the remainder we primarily consider forward slicing, even if not mentioned explicitly (i.e., we refer to it as slicing in general). Moreover, we only introduced basic concepts of PDGs, as far as needed for our slicing technique. For more details, we refer to the work of Ferrante et al. [13].

III. Motivating Example

In the following, we motivate our approach by slicing the exemplary, un-preprocessed code fragment in Figure 1. We specify the desired result of a comprehensive slice and briefly discuss possible solutions to obtain this result.

Suppose we want to slice our example without specifying whether FOO or BAR is defined. Using Line 1 as our criterion, a slice that holds for each combination of the two annotations, would consist of the following statements:

- Lines 2 and 5 are always included, since they are data-dependent on \(a\).
- Line 9–10 is always included as well due to the control dependence in Line 5.
- Line 17 is only included if BAR is not defined. Otherwise, \(b\) is redefined in Line 14 and thus prevents the dependence on Line 2.
- Line 18 is only included if FOO is not defined, otherwise the redefinition of \(c\) is not visible in the outer scope.

As we show with this example, for obtaining a comprehensive result, the corresponding slice must contain variability by means of statements that are still (virtually) annotated (here: Line 17 and 18). Particularly, this variability is expressed on the PDG and thus, indicates for which configuration a particular statement possesses data or control dependences. Such information would be omitted by existing slicing techniques, which mainly work on preprocessed programs [8].

Compared to the above-mentioned method, we could apply existing techniques only to particular variants and thus, by composing the resulting slices, obtain the same result. However, this solution comes with different limitations. First of all, this solution would not scale due to combinatorial explosion of possible variants, even for smaller systems. For instance, consider SQLite, a database engine with \(\approx 90\,000\) lines of code and more than 290 features [7]. In worst case, this results into \(2^{290} \approx 10^{87}\) possible variants to be analyzed. Additionally, we cannot conservatively approximate the slice by simply analyzing the variant in which all annotated code is included. For instance, our example result is only complete if a variant without Line 7 and 14 is analyzed, since otherwise Line 17 and 18 would not be included. Moreover, particular configuration options are mutually exclusive or may depend on each other.

Second, our proposed method can reveal additional information that is lost in preprocessed programs. Most notably, slicing programs with presence conditions enables the detection of (possibly inadvertent) feature interactions, that is, an emergent behavior caused by dependencies between features [9].

Finally, we argue that such a technique is also beneficial to overcome current limitations in analysis of variable software systems. As an example, we consider regression testing as a promising use case. Basically, by including variability information in the result (of a slice), we can identify features that are affected by a certain change. Subsequently, only variants that contain these features have to be re-tested, for instance, by using combinatorial interaction testing.

IV. Proposed Slicing Technique

In this section we present our solution to variability-aware, intra-procedural slicing. We propose an extension to the general PDG slicing approach and provide details on the actual dependence analysis we have implemented. We conclude with current limitations of our approach and future work.

A. Overall Concept

The variability of a program point imposes variability on its dependencies. For instance, in the previous section we have shown that Line 17 of Figure 1 is not always dependent on Line 2, since Line 14 would interfere with this dependence in all variants in which BAR is defined.
Consequently, we annotate the dependence of Line 17 on Line 2 with presence condition \( \neg BAR \). More generally, the overall presence condition of a dependence incorporates the presence conditions of all program aspects (e.g., particular statements, subexpressions and control-flow paths) that are required to establish this dependence. This may also require some elements to be explicitly absent\(^1\) (cf. Line 14 in the example). In order to represent variable dependencies, we make PDGs variability-aware by augmenting edges with presence conditions. More formally, we define a PDG as follows: A variability-aware PDG \((S, D)\) of a program \(P\) is a directed multigraph with nodes \(S\) and a set of labeled multi-edges \(D \subseteq S^2 \times \Phi\), where \(\Phi\) denotes the set of boolean expressions over features in \(P\). Each element of \(S\) represents a program point in \(P\). For each dependence of \(s' \in S\) on \(s \in S\) in \(P\), \(D\) contains an edge \((s, s', \phi)\), where \(\phi\) is the boolean presence condition under which this dependence exists.

As a consequence, a slice is no longer the mere set of nodes that are reachable from a criterion. We also have to compute the presence condition for each element of this set. Consequently, only considering a single path (in the PDG) to each reachable node is not sufficient, as we show in Figure 2. Using Line 3 as our criterion, we can reach Line 9 via paths \(P_1 = (3, 9)\) and \(P_2 = (3, 5, 7, 9)\). \(P_2\) is obviously not valid, as the conjunction of its presence conditions yields \(FOO \wedge \neg FOO = \bot\). Hence Line 9 may only be reached via \(P_1\) and is thus annotated with \(FOO\). Moreover, Line 7 is never reached via any valid path and is therefore excluded from the slice. Consequently, in order to obtain complete presence conditions for a slice, we aggregate the conditions of all possible paths between a criterion and a reachable statement.

In order to construct such a variability-aware PDG, we need a form of dependence analysis that can deal with variability and that yields a presence condition for each dependence. To this end, we extract control and data dependences independently, each defined as a variability-aware data-flow analysis. However, this kind of analysis inherently needs to reason about presence conditions at various points using a SAT solver (e.g., parsing, checking information gain). While in general, SAT solving is NP-complete, it is efficient enough for conditions that occur in the context of variable software systems [17].

\(^1\)For details on the general concept of this analysis technique, see Brabrand et al. [15] and also Liebig et al. [16].

B. Variability-Aware Dependence Analysis

We implement our dependence analyses on top of TypeChef, a framework for variability-aware source code analysis [10], [11], [16]. In particular, we use TypeChef to parse un-preprocessed source code, to track the scope of declarations and to analyze control flow. For our data-flow analyses, we use a variability-aware version of Monotone Frameworks, based on an implementation by Liebig et al.\(^2\)

Monotone Frameworks are a standard solution for data-flow problems [18]. In a nutshell, data-flow problems are defined as a system of equations over all program points. Each program point is assigned a pair of functions that determines the data-flow information that holds before \((in)\) and after \((out)\) its execution. Particularly, our analyses are defined in terms of constant \(gen\) and \(kill\) sets that contain so-called \(facts\). For instance, a fact may be a variable being read at a program point. For a backward analysis (as we perform for our purposes), a program point \(P\) aggregates all facts that hold at its immediate control-flow successors into \(out(P)\). In order to determine \(in(P)\), \(gen_P\) adds new facts to this set, whereas \(kill_P\) removes particular existing facts. More formally, we use equations similar to the following, where \(P\) is the program point and \(succ(P)\) is the set of all immediate control-flow successors of \(P\):

\[
in(P) = (out(P) \setminus kill_P) \cup gen_P
\]

\[
out(P) = \bigcup_{Q \in succ(P)} in(Q)
\]

The equation system is solved using a fixed-point iteration. Liebig et al. [16] extended this concept by annotating data-flow facts with presence conditions and making set operations used in the equations variability-aware. For example, they define the union on sets of data-flow facts as a pointwise boolean disjunction of their presence conditions. Thus, in order to reflect all possible variants, set membership is expressed in terms of a boolean condition (e.g., for a fact \(x\) annotated with condition \(\phi_1\) and \(\phi_2\) respectively, \(\{(x, \phi_1)\} \cup \{(x, \phi_2)\}\) yields \(\{x, \phi_1 \lor \phi_2\}\)).

Data Dependences: Using the aforementioned concept, we compute definition-use chains [13], [18], which represent the data dependences among statements. In particular, for each variable that is (re)defined in a statement, we determine the set of statements that potentially use (i.e. read) this definition. To this end, we formulate the actual data-flow problem as the set of reachable uses at each program point. A use \(v_i\) of a variable \(v\) is reachable at a program point \(P\), if there exists a path \(W\) in the control flow graph (CFG) from \(P\) to \(v_i\), such that \(W\) does not contain a redefinition of \(v\). For instance, in Figure 1 (which we use as a running example from now on), the use of \(b\) in Line 17 is reachable from Line 2 under the condition \(\neg BAR\). Moreover, since Line 2 also defines \(b\), these program points together form a definition-use chain, and we generate a data dependency in the PDG accordingly. In this analysis, \(gen_P\) contains all uses of variables at \(P\), together with their presence conditions (e.g., \(gen_{b} = \emptyset, gen_{v_2} = \{(v_2, T)\}\)). For each variable \(v\) that is defined by \(P\), \(kill_P\)

\(^2\)https://github.com/ckaestne/TypeChef/blob/aae98b55504e5e58068335491e780cb247c3326f/CRewrite/src/main/scala/de/fosd/typechef/crewrite/MonotoneFW.scala
contains all possible uses of $v$ in the program. This effectively makes all uses of $v$ after $P$ unreachable from program points before $P$, unless there is an alternative path around $P$ in the CFG. For example, $kill_2 = \{(b_{17}, T)\}$, since the only use of $b$ occurs in Line 17. However, we have to account for the condition under which a definition is associated with a particular use: As we have shown in Section III, the scope of a definition may be variable, since definitions and uses may refer to multiple declarations. For instance, Line 7 shadows the declaration of $c$ in Line 3 if $FOO$ holds. In those variants, the redefinition of $c$ in Line 9 does not affect the value printed in Line 18. Accordingly, the definition-use chain that results from combining this redefinition with the reachable use from Line 18 has the presence condition $¬FOO$. We restrict the kill of this use accordingly: $kill_9 = \{(c_{10}, T), (c_{18}, ¬FOO)\}$. More generally, we map definitions to uses as follows: Using the variability-aware type system of TYPECHEF, we resolve all declarations a definition may refer to. For each declaration, we retrieve all possible uses. Each resolved use is annotated with the presence condition of the definition and the declaration. If in some variants the declaration is hidden by declarations with a deeper scope nesting level, we additionally add the negated presence conditions of all of these inner declarations.

Control Dependences: In order to determine control dependences, we essentially compute the postdominator relation for the CFG [2], [13], again using a data-flow analysis. A node $S$ postdominates another node $T$, if every path from $T$ to the end of the control flow (usually, the function exit point) contains $S$. Thus, all nodes that do not postdominate a branching statement $B$ (e.g. a predicate of an `if` statement), yet postdominate an immediate successor of $B$, are non-transitively control dependent on $B$. Hence, for each program point, we need to determine the set of its postdominators as well as the set of reachable non-postdominators. To this end, we formulate the following analysis: For each program point $P$, there are two facts: $P_T$ denotes that $P$ is reachable, whereas $P_{⊥}$ denotes the opposite. Accordingly, $gen_P = \{P_T\}$ and $kill_P = \{P_{⊥}\}$. In other words, each program point generates its "reachable" fact and on the other hand kills its "unreachable" fact (e.g., $gen_T = \{T, FOO\}$ and $kill_T = \{T_{⊥}, FOO\}$; similar for all other program points). For all control-flow exit points $P_E$, we initialize $out(P_E)$ with a set of facts $Q_{⊥}$ for every program point $Q$. Thus, if $Q_T$ is present at some program point $P$, there exists a path from $P$ to $Q$. Moreover, if $Q_{⊥}$ is present, then there exists a path from $P$ to some exit point, such that this path does not include $Q$. Consequently, if both facts are present, $Q$ is reachable and does not postdominate $P$. In contrast, if only $Q_T$ is present, then $Q$ is a postdominator to $P$, and if only $Q_{⊥}$ is present, $Q$ is not reached in the control flow. Now, let $φ_{in}(P, F)$ be the presence condition of fact $F$ in $in(P)$, or $\bot$ if the fact is not contained in this set. Moreover, let $φ_{suc}(P, P')$ be the presence condition of the edge from $P$ to $P'$ in the CFG. Then $Q$ is control dependent on $P$ if and only if the following two conditions are satisfied:

$$φ_{in}(P, Q_{⊥}) \land φ_{in}(P, Q_T) \quad (1)$$

$$\lor_{P' \in \text{succ}(P)} φ_{suc}(P, P') \land φ_{in}(P', Q_T) \land ¬φ_{in}(P', Q_{⊥}) \quad (2)$$

The actual presence condition of this dependence is the conjunction of (1) and (2). We illustrate this analysis using Line 5 and 7: The solution for $in(5)$ contains both $(T_T, FOO)$ and $(T_{⊥}, FOO)$, since Line 7 is reachable from Line 5 in the CFG, yet Line 7 may also be skipped (depending on $a=0$). Thus, (1) evaluates to $FOO$ and accordingly Line 7 is a non-postdominator under this condition. Since (2) also evaluates to $FOO$, Line 7 is control-dependent on Line 5 in variants in which $FOO$ holds.

C. Computing a Slice

In order to compute a variability-aware slice using our augmented PDG, we first determine the subgraph that is reachable from any node in the criterion. On this subgraph, we then compute the presence condition for each node and remove nodes with unsatisfactory presence conditions. More precisely, the presence condition of a single path between a criterion and a dependent program point is the conjunction of the presence conditions of its edges. The presence condition of a sliced program point is the disjunction over all such paths.

More formally, given a PDG $G = (S, D)$ and criterion $C \subseteq S$, let $G_C = (S_C, D_C)$ be the subgraph reachable from any program point in $C$. For every $s \in S_C$, let $φ_C(s)$ denote the aggregated presence condition of $s$ in the slice defined by $C$. We define $φ_C$ as a system of equations over all $s \in S_C$ as

$$φ_C(s) = \bigvee_{(s, φ) ∈ D_C} (φ_C(s) \land φ) \lor e(s)$$

where $e(s)$ is an auxiliary function defined as

$$e(s) = \begin{cases} T & \text{if } s \in C \\ \bot & \text{otherwise.} \end{cases}$$

Thus the slice presence condition of each program point $s \in S_C$ is defined recursively as a disjunction of the slice presence conditions of each predecessor $s_p$ of $s$, in conjunction with the presence condition of the particular dependence on $s_p$. Using $e(s)$, we specify $true$ as the presence condition of all criterion nodes, basically in order to distinguish these nodes as starting points for paths and to make the solution unique. Thus all presence conditions are determined only by paths starting at criterion nodes. Because of the absorption law of propositional logic, $φ_C(s')$ effectively aggregates the presence conditions of all cycle-free paths from any $s \in C$ to $s'$. Since the set of all boolean presence conditions of a program is a finite lattice with respect to implication [15], the equation system we have specified may be embedded in a Monotone Framework. Note that here the presence conditions themselves constitute the analysis information and thus this Monotone Framework is not required to be variability-aware.

D. Limitations and Future Work

We implemented our slicing technique based on the aforementioned concepts as a prototype. Together with an example program, it is available at https://www.isf.cs.tu-bs.de/cms/team/schulze/material/varslice/index.html. While we successfully could apply our technique to simple C programs, our prototype exhibits some limitations that currently prevent us from analyzing real-world systems. In particular, our dependence analysis does not support pointers or non-scalar variable types. Hence, so far we can only analyze a small subset of C programs. Furthermore, our approach is currently limited to
intraprocedural slicing, as we ignore function calls and global variables. Finally, we do not analyze sub-statement control-flow, which makes the analysis of convoluted expressions with side-effects (e.g., conditional operator) imprecise.

Consequently, a major goal for the future is to address these shortcomings and eventually implement a variability-aware *intraprocedural* slicing technique that supports pointers and complex data types. Next, we intend to demonstrate applicability and benefits of our approach by means of empirical studies. For instance, we envision the detection of feature interactions, based on our dependence analysis, and thus, to expose possible errors. Moreover, our technique can support efficient regression testing for variable software systems based on variability-aware change impact analysis.

V. RELATED WORK

Some aspects of preprocessors in the context of program slicing have already been studied. Livadas et al. developed a technique for mapping nodes of dependence graphs to exact locations in un-preprocessed source code [19]. Furthermore, Vidács et al. proposed slicing of C++ macro expansions as an extension to slicing of C++ programs [20]. However, these approaches focus on the analysis of macros rather than the variability induced by conditional compilation. In particular, their slicing techniques are limited to preprocessed code only. Currently, we are not aware of any slicing approach that precisely analyzes conditional compilation.

However, concerning variability-aware data-flow analysis in particular, some techniques have recently been developed. Bodden et al. proposed an approach for reusing static analyses of the IFDS framework [21]. To this end, they wrap existing IFDS analyses in an IDE analysis that determines and aggregates presence conditions of data-flow facts. Similarly, Brabrand et al. provide and evaluate different ways for lifting standard data flow analysis to product lines [15]. While both approaches provide frameworks for variability-aware data-flow analysis, their current implementations restrict variability to entire statements and support only Java-based systems, which rarely use preprocessors. In contrast, Liebig et al. propose static analysis of C programs in the presence of variability, particularly control-flow and liveness analyses, and demonstrate scalability of their approach [16]. They integrated their solution into TYPEChef, which we used as the underlying infrastructure for our analyses. In particular, we adapted their variability-aware implementation of *Monotone Frameworks* to conduct the data-flow analyses presented in this work.

VI. CONCLUSION

Analyzing highly-configurable systems is a non-trivial task that is challenging due to the inherent complexity, caused by variability. In this paper, we proposed a technique for precise slicing of un-preprocessed C programs. To this end, we presented an extended PDG concept to directly represent all possible variants of a program. In order to construct a variability-aware PDG, we have adapted both, control and data dependence analysis, by integrating variability information. Moreover, we have presented an algorithm to compute a variability-aware slicing result with exact presence conditions. While our current implementation is limited regarding practical use, we laid the foundations for a useful and scalable program analysis technique for variable software systems. Our vision is to support developers and maintainers of variable systems in common tasks, such as testing or maintenance, by providing necessary information in a variability-aware manner, which would increase both, efficiency as well as scalability.

REFERENCES