Automated Software Engineering (2019) 26:417–461
https://doi.org/10.1007/s10515-019-00253-7

Change impact analysis for maintenance and evolution of variable software systems

Florian Angerer[1] · Andreas Grimmer[1] · Herbert Prähofer[2] · Paul Grünbacher[1]

Received: 14 December 2017 / Accepted: 18 January 2019 / Published online: 7 February 2019

Abstract
Understanding variability is essential to allow the configuration of software systems to diverse requirements. Variability-aware program analysis techniques have been proposed for analyzing the space of program variants. Such techniques are highly beneficial, e.g., to determine the potential impact of changes during maintenance. This article presents an interprocedural and configuration-aware change impact analysis (CIA) approach for determining the possibly impacted source code elements when changing the source code of a product family. The approach also supports engineers, who are adapting the code of specific product variants after an initial pre-configuration. The approach can be adapted to work with different variability mechanisms, it is more precise than existing CIA approaches, and it can be implemented using standard control flow and data flow analysis. We report evaluation results on the benefit and performance of the approach using industrial product lines.

Keywords Change impact analysis · Program analysis · Maintenance · Variability

This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria and KEBA AG, Austria.

✉ Paul Grünbacher
paul.gruenbacher@jku.at

Florian Angerer
florian.angerer@jku.at

Herbert Prähofer
herbert.praehofer@jku.at

1 Christian Doppler Laboratory MEVSS, Institute for Software Systems Engineering, Johannes Kepler University Linz, Linz, Austria

2 Institute for System Software, Johannes Kepler University Linz, Linz, Austria

1 Introduction

Variability is a property of software systems allowing their customization to different application scenarios. It becomes essential when developers have to implement similar solutions to meet a wide range of customer requirements (Lettner et al. 2014b). Support for variability is regarded as a successful strategy for increasing software reuse and for developing highly customized solutions (Svahnberg et al. 2005). Dealing with variability, however, leads to many challenges when developing, maintaining, testing, or analyzing systems. For instance, developers need to ensure that different software variants behave as expected. However, it has been shown that analyzing all possible system variants is computationally infeasible, even for small systems (Liebig et al. 2013). Research on variable software systems has progressed significantly: for instance, researchers in software product lines and feature-oriented software development have developed family approaches (Thüm et al. 2014) that allow analyzing the whole space of software variants by exploiting commonalities between variants. Such approaches have been shown to be very effective, particularly for program analysis (Liebig et al. 2013). Often, the existing approaches assume that the source code is annotated directly with variability information, which is the case, e.g., in annotation-based product lines that use preprocessors (Kästner et al. 2011). However, other variability mechanisms, such as load-time configuration options, play an equally important role and analysis support is also needed for such cases.

An important application area for program analysis is change impact analysis (CIA), i.e., the identification of the potential consequences of a change, or the estimation of what needs to be modified to accomplish a change (Arnold 1996). Our aim is to improve support for CIA of variable software systems, an area of high practical relevance, particularly for clone-and-own product lines (Linsbauer et al. 2014; Rubin and Chechik 2013). In this paper, we thus present an interprocedural and configuration-aware CIA approach that uses and propagates variability information. Our approach can handle load-time configuration options representing software variability. While some program analysis approaches (e.g., Hammer et al. 2006) can handle such runtime variability by attaching path conditions to system dependence graphs (SDGs) to improve the precision of slices, their scalability to large programs is limited, as path conditions need to be extracted for nearly every conditional statement. Our approach thus uses a *conditional system dependence graph (CSDG)* (Angerer et al. 2014), an extended representation of an SDG.

We demonstrate the benefits of our approach using two use cases derived from an analysis of development practices of our industry partner (Lettner et al. 2014b, a). We illustrate how the approach facilitates *development and maintenance in domain engineering* by supporting software engineers that need to determine the impact of changing a set of source code elements of a product family. Specifically, we demonstrate how our configuration-aware CIA allows to automatically determine the set of possibly impacted products. Such analysis has major benefits for software evolution: for instance, it allows reducing regression testing to the affected product variants only. It further simplifies software deployment, as updates only need to be rolled out to customers affected by certain changes. We further show how our approach supports *development and maintenance in application engineering* by supporting software engineers changing a specific product variant, which needs to be adapted after deriving it from a product line. This is a frequent case in clone-and-own product lines (Linsbauer et al. 2014; Rubin and Chechik 2013), when engineers customize and extend the software to the specific requirements of customers. Again, manual CIA would be error-prone and infeasible due to the high configurability of many large systems in this case.

Our interprocedural CIA approach annotates potentially impacted elements with variability information to determine the set of affected products after a change. It provides the following benefits compared to existing techniques: (i) *the approach can be adapted to work with different variability mechanisms.* Our technique does not assume that the source code is directly annotated with variability information, i.e., unlike existing approaches it does not assume an annotation-based mechanism that is resolved at compile-time. Instead, it can handle configuration options that remain constant after being loaded during the startup of an application. Although the approach and application focuses on load-time variability mechanisms, we have shown in Angerer et al. (2017) that it can also handle annotation-based variability. (ii) *The approach provides more precise results than existing CIA.* Specifically, it discovers contradicting product configurations and determines source code that can never be executed. This allows reducing the size of the change impact. (iii) *The approach uses standard control flow and data flow analyses.* We ease the implementation of the approach in different contexts by avoiding the use of new control and data flow analyses.

This article is based on an earlier conference paper (Angerer et al. 2015). Compared to this publication our article provides a detailed description of the conditional system dependence graph including its formal definition, the description of a tool-supported method supporting key uses cases to interactively explore and investigate the SDG, and a detailed description of the technical foundations for the conditional change impact analysis, including a description of the algorithms. We further discuss the background literature in more detail.

Specifically, our paper is organized as follows: Sect. 2 discusses background literature relevant for our work. Section 3 illustrates the research problem using a small example. Section 4 explains the conditional system dependence graph (CSDG) providing the foundation for our analyses, and presents a tool-supported method to interactively explore the SDG. Section 5 presents our configuration-aware CIA approach and its implementation. Section 6 shows the results of evaluating the approach regarding its benefit and performance. Section 7 presents related work. Section 8 rounds out the paper with a conclusion and an outlook on future work.

2 Background

We discuss background literature from the areas of software product lines, variability mechanisms, static program analysis, and variability-aware program analysis.

2.1 Software product lines

A software product line (SPL) (Czarnecki and Eisenecker 2000; Pohl et al. 2005) has been defined as *a set of software-intensive systems sharing common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* (Clements and Northrop 2001). SPLs identify and manage common functionality relevant for most customers and then support deriving customized solutions based on the customers' individual requirements. The SPL approach consists of two life cycle phases (Pohl et al. 2005): in domain engineering, the commonality and variability of the SPL is defined and reusable artifacts are developed. In application engineering, a concrete application satisfying the specific customer requirements is realized by exploiting the SPL's commonality and variability. Compositional approaches (Kästner et al. 2008) provide an automated process for composing new product variants by selecting the software artifacts, e.g. features (Apel and Kästner 2009), aspects (Kiczales et al. 1997), or deltas (Schaefer et al. 2010), to be included.

2.2 Variability mechanisms

Numerous variability mechanisms exist that allow to handle different product-specific implementations. A common practice is to have a common base and just vary the implementation where it is necessary (Clements and Northrop 2001). For example, developers use preprocessor directives (Liebig et al. 2010), custom-developed configurators (Lettner et al. 2013), aspect-oriented programming (Kiczales et al. 1997), delta-oriented programming (Schaefer et al. 2010), feature-oriented programming (Apel and Kästner 2009), or load-time configuration options (Lillack et al. 2017) to name but a few.

For example, the C preprocessor (CPP) is a popular tool which adds an additional step to prepare the source code for the compiler (Kernighan and Ritchie 1988). The CPP also provides a directive that allows to include or exclude source text depending on a given configuration. Therefore, it is a simple but powerful mechanism for implementing variability (Liebig et al. 2010). However, the use of preprocessors also has major drawbacks. For example, the CPP introduces directives that also appear in the source code, i.e., the CPP defines a language for metaprogramming (Liebig et al. 2010). Therefore, analysis tools for the C programming language can only work on processed source code or need to understand CPP directives (Kästner et al. 2011). Furthermore, while CPP is well integrated into the majority of the C compiler tool chains (e.g., the GNU Compiler Collection), other preprocessors are not related to a programming language and the integration must be done manually (Kästner 2012).

Another widespread technique for implementing variability are configuration options that are loaded from a file or provided as program arguments to control conditional execution (Lillack et al. 2014). Load-time configuration options can be seen as a way between compile-time and run-time configuration. Syntactically, they are indistinguishable from run-time configuration because values are loaded from a source and then stored in program variables. Semantically, load-time configuration options are close to compile-time constants because the value is loaded in the startup phase of the program and remains constant over program execution time. This has the advantage that no additional techniques are required and variability can directly be implemented in the programming language.

2.3 Static program analysis

Static program analysis techniques acquire information about the structure but also run-time behavior of a program without executing it (Nielson et al. 1999). This allows improving the quality of code, e.g., by providing means to reveal inadequate code constructs ("code smells"), violations of programming guidelines, and potential defects (Louridas 2006). Well-known static analyses are control flow analysis, data flow analysis (Nielson et al. 1999), abstract interpretation (Cousot and Cousot 1977), program slicing (Weiser 1981), and change impact analysis (Arnold 1996). Control-flow and data-flow analyses were originally developed in context of compiler technology (Muchnick 1997). Control-flow analysis determines how the program can be executed, i.e., which program paths may occur. Data-flow analysis determines the values which can be created in a program and how they can be manipulated and used (Allen and Kennedy 2001). Typical examples of data-flow analysis methods are reaching definitions, live variables, and available expressions.

Program slicing is a technique for reducing a program to the subset of statements (the slice), which faithfully represents a specific program behavior (Weiser 1981). The method is based on the observation that for producing a particular program behavior, often only a subset of a program is required. The goal usually is to reduce the effort required to understand and maintain the program by only having to consider a part of it. The original slicing algorithm by Weiser (1981) uses the control flow graph of a program. Ottenstein and Ottenstein (1984) then formulated slicing as a graph reachability problem on the program dependence graph. However, this method was limited to intraprocedural slicing. Furthermore, Horwitz et al. (1990) introduced the SDG and a traversing algorithm for finding interprocedural slices. In general, using reachability algorithms on dependence graphs is the most popular way for computing slices (Xu et al. 2005).

Change impact analysis (CIA) is the process of determining the potential effects of a proposed modification in the software (Bohner 2002). For instance, in the context of this paper this means to determine a set of impacted statements after modifying source code. Since the exact impact of a change is again hard or impossible to compute, CIA approaches just compute a possible impact. CIA approaches can be classified into guided and unguided techniques, techniques using heuristics, approaches using static or dynamic approaches, and combinations of several approaches.

2.4 Variability-aware program analysis

Using program analysis techniques during development and maintenance of SPLs requires to consider all possible product variants (Brabrand et al. 2012). This is necessary because every single possible product variant must be covered to ensure that the whole SPL is correct. Unfortunately, the number of possible product variants grows exponentially with the number of available options (Thüm et al. 2014). Therefore, researchers developed variability-aware program analysis that use the same principle as SPLs for analyzing SPLs (Brabrand et al. 2012; Liebig et al. 2013). Such approaches perform the analysis for all common parts just once and only diverge the analysis if the program diverges. The core concepts of variability-aware program analysis are late splitting and early joining (Liebig et al. 2013). Late splitting means that the analysis is performed without variability until variability is encountered. Early joining is the concept of collapsing intermediate analysis results as soon as possible, i.e., if data flow from different product variants reaches a destination that is common for several product variants again. Variational data structures efficiently represent variability in data and enable variability-aware computations (Walkingshaw et al. 2014). For example, managing the artifacts of a SPL requires information about when to include the artifacts in a product. Variational data structures are often used for variability-aware program analysis. A well known example of such data structure is a variational AST. TypeChef, for instance, parses preprocessor-annotated source code and represents the variability of the source using a variational AST (Kästner et al. 2011). The CSDG presented in Sect. 4 is another example of such a variational data structure.

3 Problem illustration

Change impact analysis allows to automatically determine and systematically review the possibly impacted source code for changes. However, state-of-the-art CIA techniques (Chen and Rajich 2001; Jász et al. 2008; Bohner 2002; Black 2001) do not consider load-time variability. For example, Listing 1 shows an illustrative program example that can be configured by enabling or disabling the configuration options `c0` and `c1`. Existing CIA techniques do not provide information about the product variants that are affected by a change. Even this small configurable program shows that manually determining the set of affected products is difficult due to many dependencies as can be seen in the simplified SDG for Listing 1 shown in Fig. 1. Assume a developer changing the return statement in line 38, represented as a node in the lower-right corner of the SDG. Existing CIA techniques follow the forward edges and mark all visited nodes as possibly impacted by the intended change, i.e., the nodes labeled `return d.bar()* d.bar()`, `return 2 * d.bar()`, `int res = obj.foo(d)` and `System.out.println(res)` are in the set of impacted statements.

Performing configuration-aware analysis is much harder if also considering load-time variability, as one needs to find out if the statements are executed and the data flows are valid under certain conditions. For example, the statements `object = new B(); return;` and `object = new C();` depend on the conditions $c0$ and $\neg c0 \wedge c1$. Therefore, the bodies of classes `B` and `C` also depend on these conditions. In class `D`, an additional local condition $\neg c1$ appears. Hence, statement `return 1;` is only executed if the condition $(c0 \vee \neg c0 \wedge c1) \wedge \neg c1$ holds, while statement `System.out.println(res)` is executed if $\neg c0$ holds. However, combining and simplifying both conditions results in $false$, i.e., changing the statement `return 1;` has no impact on `System.out.println(res)`.

```
class Main {
  static Properties p = Properties.load("conf.prop")
  static boolean c0 = "on".equals(p.getProperty("c0"));
  static boolean c1 = "on".equals(p.getProperty("c1"));

  public static void main(String[] args) {
    A obj = new A();
    D d = new D();
    if(c0) {
      obj = new B();
      return;
    } else if(c1) {
      obj = new C();
    }
    int res = obj.foo(d);
    System.out.println(res);
  }
}

class A {
  int foo(D d) {
    return 2;
  }
}
class B extends A {
  int foo(D d) {
    return d.bar() * d.bar();
  }
}
class C extends A {
  int foo(D d) {
    return 2 * d.bar();
  }
}
class D {
  int bar() {
    if(!c1) return 1;
      return 0;
  }
}
```

Listing 1 Small configurable program

Researchers have already developed program analysis techniques considering the variability of programs, e.g., when performing data flow analysis (Liebig et al. 2013). However, many current techniques assume that source code is annotated with variability information, that is resolved during compilation by only considering the syntactic structure of a program. However, as pointed out, many product lines and configurable software systems use different variability mechanisms, such as load-time configuration options. There are some approaches, e.g., (Snelting 1996), that are able to consider such runtime variability, but they do not differ between configurations that remain constant throughout execution and the regular control flow of a program. In the example

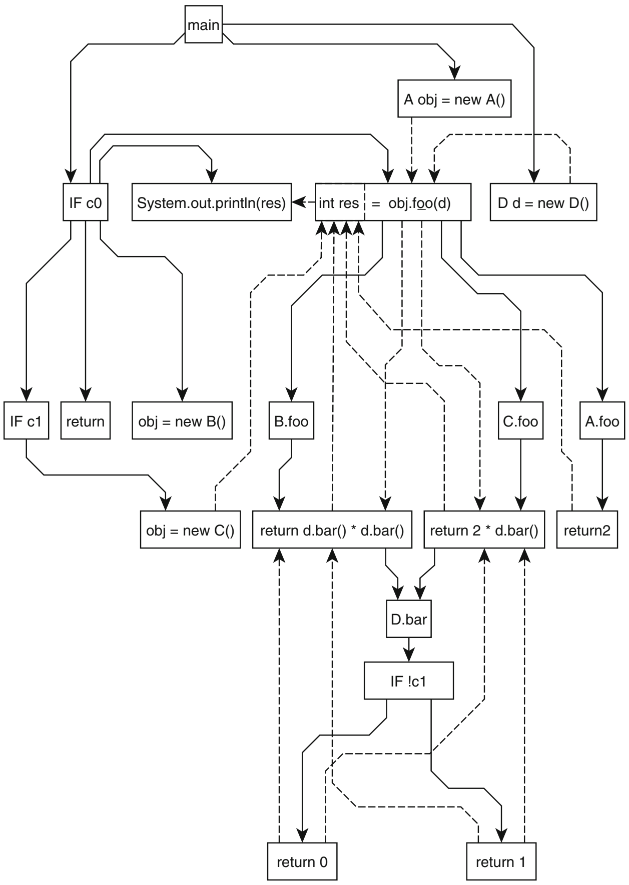


Fig. 1 SDG for the small configurable program in Listing 1

in Listing 2, the value of a configuration option is stored in a local variable (line 5), which is used subsequently to decide which other configuration option to load (line 7). Current analysis techniques assume a strict separation of the variability mechanism and program control flow. Hence they cannot handle such situations, i.e., they do not recognize that the execution of the statements in lines 15 and 17 also depends on the configuration option `c0`. In this small example it is obvious that the execution of the statements in lines 15 and 17 also depends on conditions `c0` and `c1`. Finally, the call in line 19 also depends on both configuration options since the reaching objects depend on those configuration options.

4 Conditional system dependence graph

Change impact analysis (CIA) is commonly performed by following edges in a system dependence graph (SDG) (Horwitz et al. 1990). An SDG consists of nodes representing concrete and abstract program elements and edges encoding control and data dependencies. To make the CIA configuration-aware, the variability of the program

```
class Main {
  static Properties prop = Properties.load("conf.prop")

  public static void main(String[] args) {
    boolean c0 = "on".equals(prop.getProperty("F"));
    boolean c1;
    if (c0) {
      c1 = "on".equals(prop.getProperty("X"));
    } else {
      c1 = "on".equals(prop.getProperty("Y"));
    }

    A obj;
    if (c1) {
      obj = new A1(); // indirectly depends on c0
    } else {
      obj = new A2(); // indirectly depends on c0
    }
    obj.foo();
  }
}
```

Listing 2 Influence of configuration options on program execution

must be represented in the SDG. In this work, we use a *conditional system dependence graph* (CSDG), an extension of the SDG that represents variability in form of presence conditions. The presence conditions encode if a dependency exists in a configuration. Thus, a configuration-aware CIA can be performed that also considers the different product configurations.

4.1 System dependence graph

An SDG is a directed graph representing different kinds of dependencies between program elements. It usually represents control-flow and data-flow dependencies, but other types such as definition-use dependencies are also possible (Horwitz et al. 1990). In this work the following node types are used for representing concrete or abstract program elements in the SDG (cf. Fig. 2 showing the SDG of the program from Listing 3).

Method nodes represent methods, procedures, functions or any other type of callable program unit. In the following we make no distinction between those types of program elements, but use the term *method* as a synonym for all types of callable program units. Thus, method nodes represent the entries of callable program elements. Further, they group parameter nodes and statement nodes. Figure 2 contains method nodes for the procedures `main`, `foo` and the two variants of `bar` from classes `M1` and `M2` from Listing 3. Additionally, the `main` node also acts as a container for global variables.

Statement nodes represent the statements in methods and always belong to one method node.

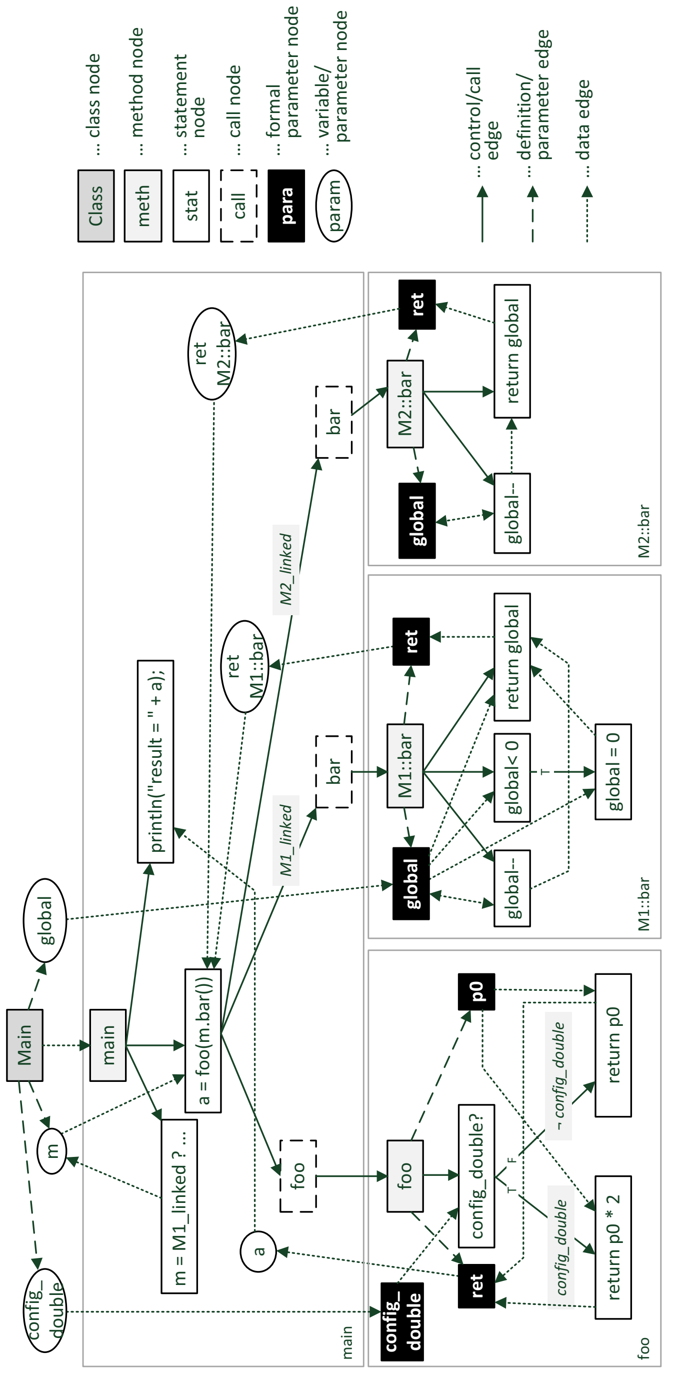


Fig. 2 The CSDG for the sample program in Listing 1

```
class Main {
  static Properties p = Properties.load("conf.prop")
  static boolean M1_linked =
        "M1".equals(p.getProperty("m"));
  static boolean M2_linked =
        "M2".equals(p.getProperty("m"));
  static boolean config_double =
        "on".equals(p.getProperty("config_double"));
  public static int global = 0;
  static M m;

  public static void main(String[] args) {
    m = (M1_linked) ? new M1() : new M2();
    int a = foo(m.bar());
    print("result = " + a);
  }
  static int foo(int p0) {
    if (config_double) {
      return p0 * 2;
    } else {
      return p0;
    }
  }

  static interface M {
    public int bar();
  }

  static class M1 implements M {
    public int bar() {
      global --;
      if (global < 0) {
        global = 0;
      }
      return global ;
    }
  }

  static class M2 implements M {
    public int bar() {
      global --;
      return global ;
    }
  }
}
```

Listing 3 Sample program for illustrating the CSDG

Call nodes are used to represent method calls, i.e., call nodes are special types of statement nodes. They are particularly important in the SDG as they link statement nodes to method nodes. Figure 2 shows call nodes for the calls to methods `foo` and for both versions of `bar`.

Formal parameter nodes are used for representing any data dependencies between a method and its environment. The return values of methods and access to global variables are also represented as formal parameter nodes. Figure 2 contains formal parameter nodes (with black background) for the method parameters, return values, and the global variable `global` accessed in the two versions of procedure `bar`.

Actual parameter nodes represent expressions for actual parameters in method calls.

The SDG contains the following types of edges for representing the various types of dependencies between program elements:

Control edges link method nodes with statement and call nodes directly contained in the method. Control edges represent that a statement or call will be executed if the method is called. Furthermore, conditional statements are represented as control edges labeled with true (T), to connect the condition with all the nodes of the then branch, and edges labeled with false (F), to connect the condition with the nodes of the else branch. For example, the statement `a := foo(bar())` in Listing 3 is executed unconditionally as soon as the procedure `main` is executed. Therefore, the corresponding statement node in the SDG is directly connected to the corresponding method node with a control edge. In distinction, the statement `RETURN p0 * 2` is only executed if the condition `config_double` evaluates to true. This is represented by the control edge labeled with `T` between the corresponding statement nodes in the SDG. On the other hand, the statement `RETURN p0` is only executed if the same condition evaluates to false. This is represented by the control edge labeled with `F`. Control edges are shown as solid lines in Fig. 2.

Parameter edges connect method nodes with all formal parameter nodes used by the method. Analogously, there are parameter edges from a call node to the actual parameter nodes. Parameter edges are represented with dashed-dotted lines in Fig. 2.

Data dependence edges show data dependencies between parameter or variable nodes and statement nodes in the SDG. Data dependence edges between actual and formal parameter nodes show parameters passing during method calls. Data dependence edges between statement nodes represent data-flow dependencies. In Fig. 2 data dependencies are shown as dashed lines.

4.2 Presence conditions

A CSDG is built from the SDG by annotating edges with presence conditions. There are different ways how variability can be implemented in a program and the presence conditions are used for expressing and abstracting from different variability implementation concepts. For example, variability can be implemented using conditional statements testing configuration settings (i.e., load-time configuration options (Lillack et al. 2014, 2017)), or module link configurations determine the modules to be used in case alternative modules are available. Furthermore, compile-time variability (Kästner et al. 2011) allows implementing variability by selecting and composing source code snippets. Such approaches may, however, modify the source code in a non-structured way, which can be prevented by transforming compile-time to load-time variability (von Rhein et al. 2016).

In the CSDG, presence conditions are encoded as Boolean formulas representing the set of valid product variants (cf. Kästner et al. 2011). Thus, an edge is enabled for a specific configuration if the condition is satisfied with respect to that concrete product configuration.

Dependent on the type of edge, the presence conditions can be interpreted as follows: if a control dependence edge is annotated with a presence condition, the execution of the program element will only occur in a configuration satisfying the presence condition. Analogously, a presence condition attached to a data dependence edge means that the data flow only exists in the product variants satisfying the presence condition.

Consider the configuration variable `config_double` in Listing 3: There will be presence conditions on the edges connecting the statement node `config_double` in procedure `foo` with nodes `RETURN p0*2` and `RETURN p0` as shown in the CSDG in Fig. 2. Further, the classes `M1` and `M2` contain two alternative implementations of procedure `bar`. Depending on which class is actually instantiated, one of the variants will be called. Therefore, there are presence conditions `M1_linked` and `M2_linked` on the two respective edges.

4.3 Formal definition of the CSDG

After getting an intuitive understanding, we give a formal definition of the CSDG in the following (cf. Ferrante et al. (1987)). First, let us formally define an SDG.

Definition 1 **SDG.** An SDG is a tuple

$$SDG = (V, E, T)$$

where V is the set of nodes of the SDG and $E \subseteq V \times V$ are the edges and $T : E \rightarrow \{ctrl, data\}$ is a function determing if the edge represents a control or data dependency.

The control-flow edges $e = (a, b) \in E \wedge T(e) = ctrl$ represent the execution semantics. Intuitively, a node b is control dependent on a node a if the node a *decides* if b will be executed.

The definition of control dependency is based on Ferrante et al. (1987):

Definition 2 (*Control dependency*) There is a control dependency $e = (a, b) \in E, T(e) = ctrl$ between nodes $a, b \in V$ if node a decides if node b is executed. We write $a \xrightarrow{ctrl} b$. More formally: $a \xrightarrow{ctrl} b$ if

1. There is a path p from a to b in the control-flow graph such that b post-dominates every vertex $v \in p, v \neq a$, and
2. b is not the immediate post-dominator of a

This definition uses the post-dominator relationship on the CFG which is defined as follows.

Definition 3 (*Post-Dominator*) Node b *post-dominates* node a in the CFG if every path from a to $EXIT$ (excluding a) contains b.

In other words, if a node has children in the control flow graph (CFG), then the node is a decision point. If a node b post-dominates a child of node a, then node b is control dependent on node a. In this case, node a then was the nearest decision point for the execution of b.

Data dependence edges are determined by the reaching definition relation between nodes.

Definition 4 (*Data dependency*) There is a data dependency $e = (a, b) \in E, T(e) = data$ between nodes $a, b \in V$ if node a defines the value of some data element and this definition is read in b. We write $a \xrightarrow{data} b$.

Based on the definition of the SDG, a formal definition of the CSDG is as follows.

Definition 5 **CSDG.** A CSDG is a tuple

$$CSDG = (V, E, T, C, PC)$$

where V, E and T are the same as in the SDG, C is a set of Boolean configuration variables and PC is the presence condition function

$$PC : E \rightarrow (\mathcal{B}^C \rightarrow \mathcal{B})$$

where $\mathcal{B} = \{True, False\}$ is the set of Boolean values and which for each edge gives a function that determines a Boolean presence value based on a vector of values of Boolean configuration variables C. Note, that the presence condition function will be constantly $True$, if the edge is not dependent on a configuration.

4.4 Support for exploring the CSDG

The CSDGs of real-world programs get huge as our evaluation in Sect. 6 will show. Therefore, tool support is needed to convey the program dependencies and the variability information encoded in the CSDG to developers.

The SDG Browser (Feichtinger 2017) is a tool for visualizing and interactively browsing the dependencies of a program. It allows selectively displaying nodes of corresponding program elements and following their dependencies. Specifically, it covers four use cases addressing fundamental development and maintenance activities: *executions*, *variable assignments*, *change impact*, and *change cause*.

Executions support finding out how a statement (or procedure) can be executed, i.e., it reveals all possible program paths leading to its execution. A developer can select a statement in an editor and then start the SDG Browser showing the respective CDSG node for this statement. The developer can then follow the control dependence edges backward, to visualize all the program paths leading to the selected statement. This effectively gives a graph which contains all possible stack traces that could be observed for any execution.

For example, Fig. 3a shows the possible execution paths for the statement `return 0`. Starting from `return 0`, the control paths are followed backward, until the node `main` is reached.

Variable assignments allow to find out how a specific variable can be set. In this use case a developer first selects a variable in the editor, which is then displayed as the initial node in the SDG Browser. If multiple instances of this variable exist, e.g., in case of an object variable, all its instantiations are displayed. Based on the data dependencies, all assignment statements affecting this variable are accessed. From the assignment statements the executions as outlined above can be shown. Thus, the use case shows how the assignment statements to the selected variable can be executed and the variable can be set.

Figure 3b shows the execution of the assignments to variable `obj`. The variable declaration in the CSDG has a data dependency edge to the three assignment statements (not shown in the diagram). From those the control edges are followed backwards.

Change impacts can be computed to support a developer in determining statements or variables possibly affected by changing or executing a particular statement or variable. This feature allows displaying a sub-graph of the CSDG, which is connected by control or data dependencies. Therefore, after selecting a program element or a group of program elements, the control as well as data dependence edges are followed forward. The user can selectively open parts of the sub-graph for inspection.

Figure 4a shows the change impact of statement `return 0`. Starting with this statement, the algorithm follows all data and control dependency edges. If the developer changes, e.g., the returned value, there might be an impact on the selected statements.

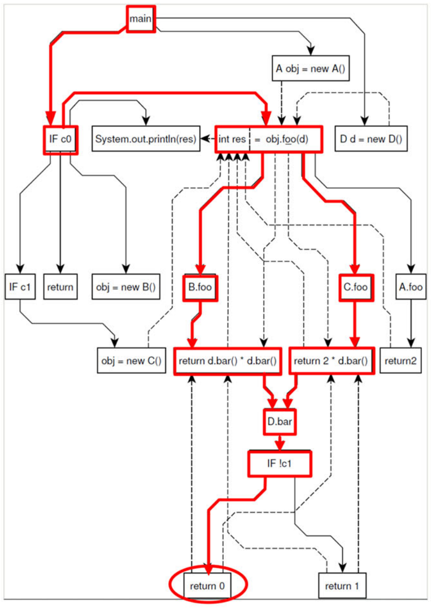
Change cause is inverse to analysing the change impact. This feature determines for any program element the statements or variables which might have an impact on its value or execution. It works similar to change impact, but follows the control and data dependencies in a backward direction. This use case can be helpful, for instance, if a developer wants to know the cause that the execution of a statement changed.

Figure 4b shows the change cause of statement `return d.bar() * d.bar()`. The return value can be influenced by the two returns of method `d.bar`. However, also the execution of the predecessor control node `B.foo` might effect the execution, i.e., if it is executed or not. Continuing from those nodes, all the control and data dependencies are followed backwards.

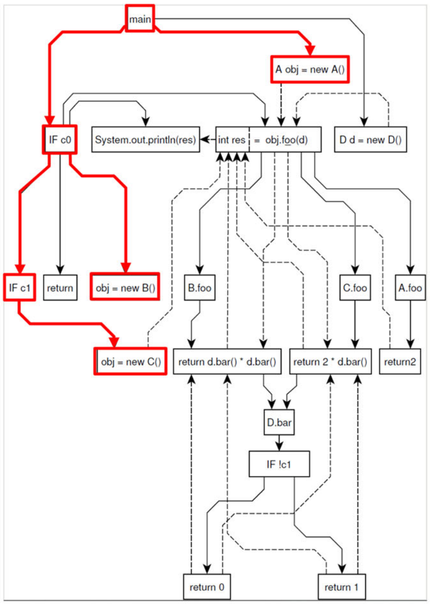
SDG browser tool

Figure 5 shows a screenshot of the SDG Browser tool, currently showing the variable assignment for a variable `tmp`. In the example program, there are two instances of the building block declaring the variable, therefore two instances of the variable exist. Those are set in different ways. The tool allows expanding/collapsing nodes and following edges (+ button). In the example, one execution is partially expanded,

Fig. 3 The use cases for using the SDG

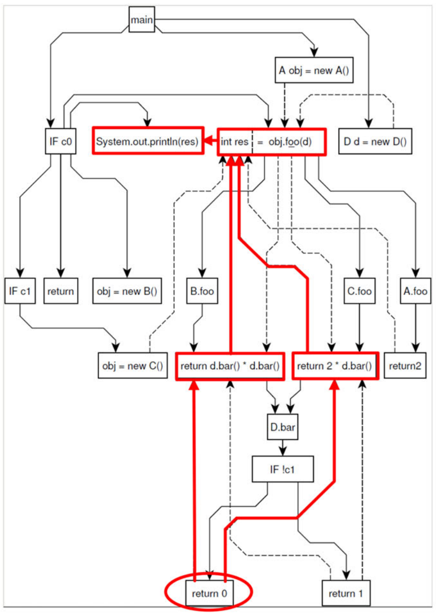


(a) Showing execution paths for statement `return 0`.

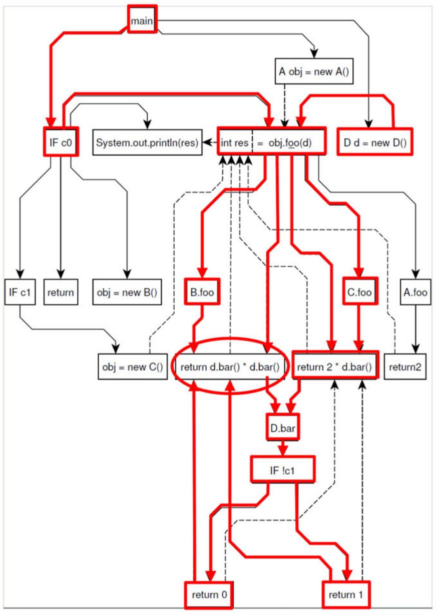


(b) Shows the possible assignments and their execution paths for variable `obj`.

Fig. 4 The use cases continued from Fig. 3



(a) Shows the change impact of statement `return 0`.



(b) Shows the change cause of statement `return 0`.

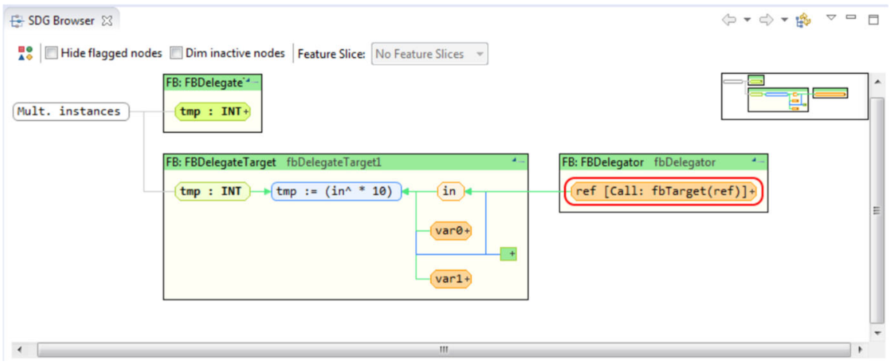


Fig. 5 Screenshot of the SDG Browser tool showing a variable assignments for two instances of a variable `tmp`

the other collapsed. The statement nodes are contained in the procedure nodes (green boxes).

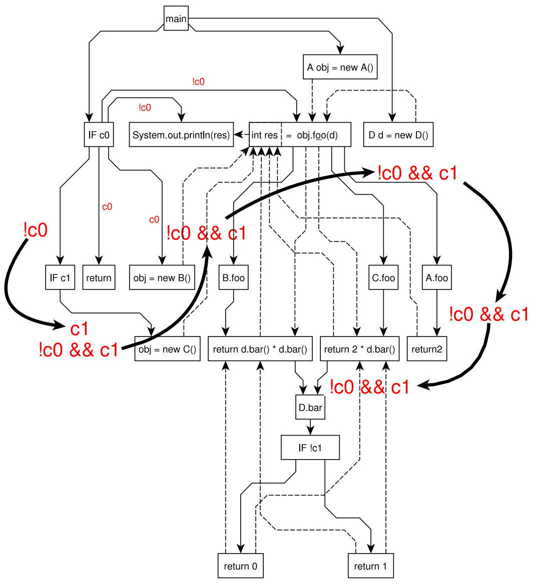
Considering configuration options

In the use cases discussed above, no configuration options are considered. When considering a specific configuration setting, certain edges are not enabled and therefore the edges will be excluded in the use cases. This is straightforward for the executions and variable assignment use cases. For example, if we know that option `c1` is not set, then the control edge from `return 0` to the statement `if (!c1)` is not enabled and the return statement cannot executed. In the SDG Browser tool the selection “Dim inactive nodes” allows shadowing the nodes which cannot be executed due to the current configuration. However, for change impact analysis configuration options propagate. This is discussed in detail in Sect. 5.

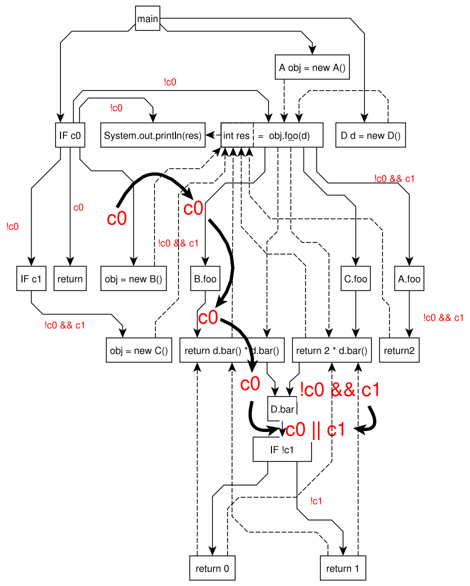
5 Configuration-aware change impact analysis

Recall from Sect. 3 that configuration options influence the control and data flow in the program and therefore need to be considered when determining a change impact. We thus now show how the CSDG with the presence conditions is used to propagate configuration options to see if and under which conditions a change impact really prevails. The propagation algorithm starts at the presence conditions from the CSDG and propagates and combines them along the forward slices for finally determining a configuration condition for the change impact. We call these propagated conditions *impact conditions*, as they represent the influence of configuration options on the change impact. Further, configuration conditions may spread in the program by assignments along data dependencies. The approach therefore computes reaching definitions for configuration conditions to know for each program position which variables store which configuration conditions. We call this mapping of variables to configuration conditions *reaching conditions*.

Fig. 6 The steps for the propagation of the variability information showing the CSDG and how the presence conditions are propagated

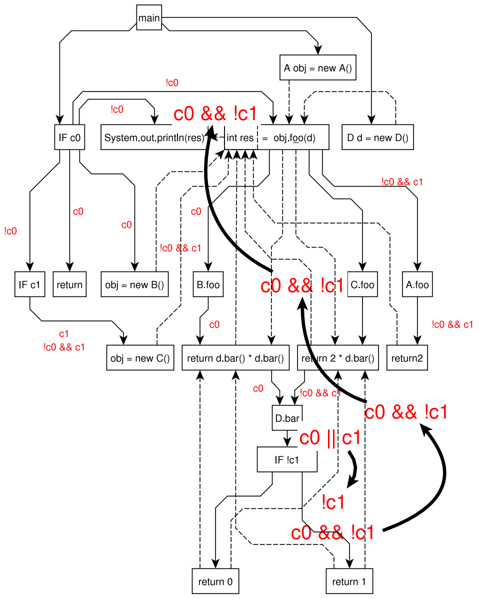


(a) Snapshot (a): Propagating presence condition `!c0` to change impact criterion.

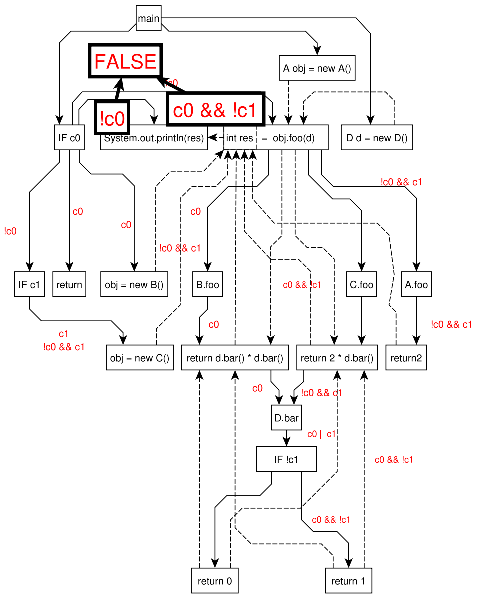


(b) Snapshot (b): Propagating presence condition `c0` to change impact criterion.

Fig. 7 The steps for the propagation continued from Fig. 6



(a) Snapshot (c): Propagating presence condition `!c0 && c1` from change impact criterion along forward dependencies.



(b) Snapshot (d): Combining presence condition of change impact criterion and of impacted statement to false.

5.1 Problem illustration

We now explain condition propagation using an example. Figures 6 and 7 illustrates how the variability information is propagated to compute the impact conditions for the introductory example from Sect. 3. Let's assume that the statement `return 1` on line 37 is to be modified and the impact of this change needs to be computed.

Snapshot (a) First, as shown in Fig. 6a, the initial presence condition `!c0` arises from statement `IF c0` which tests a configuration value. Since the successor `IF c1` is in the else branch of the if statement, the condition `c0` is negated finally revealing the impact condition `!c0`. The first propagation step moved condition `!c0` over node `IF c1` and reaches the next presence condition `c1`. In this case, the two conditions are conjunctively combined resulting in `!c0 && c1`. The resulting impact condition is then moved iteratively to successors `int res = obj.foo(d)`, `C.foo`, `return 2*d.bar()`, and `D.bar`. Since these nodes are not related to configuration, the propagated condition does not change and the impact condition of this ingoing edge to `D.bar` is `!c0 && c1`.

Snapshot (b) Figure 6b shows the next five propagation steps. The presence condition `c0` from the edge to the node representing the then-branch is moved iteratively to successors `int res = obj.foo(d)`, `B.foo`, `return d.bar()*d.bar()`, and `D.bar`. Again, these nodes are not related to configuration and thus do not modify the condition. However, when the condition is moved over node `D.bar`, it is disjunctively combined with condition `!c0 && c1` because the node is a method node and the two incoming edges represent call edges. Therefore, the conditions are disjunctively combined because either of the calls may happen and the analysis has to assume that both calls are possible. This node is thus called a join node and the two incoming conditions are combined to `c0 || c1` which defines the impact condition for the outgoing edge of `D.bar`.

Snapshot (c) Figure 7a shows the final propagation steps and the steps for the change impact analysis. Condition `c0 || c1` from node `D.bar` first propagates over node `IF !c1` and results in impact condition `(c0 || c1) && !c1` which is equal to `c0 && !c1`. Now, the propagated condition is at the change impact criterion `return 1` and the change impact is computed by following the outbound edges while carrying the already propagated impact condition.

Snapshot (d) Finally, the print statement is reached (cf. Fig. 7b) and the impact condition `c0 && !c1` is combined with the incoming condition `!c0`. The combination is a contradiction, i.e., the result is always false and we can conclude that there will be no change impact to the print statement when changing `return 1`.

Let's now consider the spreading of configuration conditions by assignments along the data dependencies to intermediate variables. For example, in Listing 2 the value of a configuration option is stored in a local variable `c0` (line 5). which is used subsequently to decide which other configuration option to load (line 7). As pointed out above, currently available analysis techniques cannot handle such situations as they assume a strict separation of the variability mechanism from program control flow, i.e., they do not consider that the execution of the statements in lines 15 and 17 also depends on the configuration option `c0`. In this small example it is obvious that the above condition reaches the statements in lines 15 and 17 and thus their execution also depends on conditions `c0 && c1` and `!c0 && !c1`. Finally, the call in line 19 also depends on both configuration options since both configuration options reach this statement.

It is thus essential to know which configuration conditions can reach which locations in the program. This is accomplished by a reaching definitions calculation for configuration variables, which we denote as reaching conditions.

5.2 Definitions

Relating to the formal definition of the CSDG (cf. Sect. 4.3) and for formally representing the impact conditions and reaching conditions resulting from the propagation approach, we introduce the *impact conditions* function and *reaching conditions* function as follows:

Definition 6 Impact Conditions (IC) is a function

$$IC : E \rightarrow (\mathcal{B}^C \rightarrow \mathcal{B})$$

which maps edges in E to functions mapping a vector of values of configuration variables C to a Boolean impact condition value.

That means IC determines if the edge is enabled in the current propagation and the given configuration settings. Note, that impact conditions and presence conditions have equal structure.

Definition 7 Reaching Conditions (RC) is a function

$$RC : E \rightarrow (Var \rightarrow (2^{\mathcal{B}^C \rightarrow \mathcal{B}}))$$

which maps edges in E to functions from program variables $v \in Var$ to a (possibly empty) set of configuration conditions, where Var is the set of program variables used for configuration conditions. That means RC gives for an edge and a variable the configuration conditions which can reach this edge.

In summary, we will use the following terms when further explaining our approach:

- *Presence conditions* (PC) as introduced in Sect. 4.2 represent the initial conditions on edges in the CSDG resulting from configuration options.
- *Impact conditions* (IC) are the conditions resulting from the propagation and combination of conditions along the edges in the CSDG. Impact conditions relate to presence conditions, as presence conditions are the starting point for propagation.
- *Reaching conditions* (RC) are the configuration conditions stored in variables. Reaching conditions are computed for edges in the CSDG by computing reaching definitions sets for configuration variables in the propagation process.
- The *change impact criterion* is that program element for which the change impact is computed.

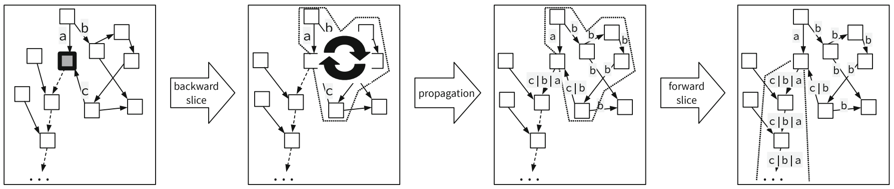


Fig. 8 Overview of the CA-CIA approach. Starting with the CSDG containing the presence conditions, first a backward slice is computed from the CIA criterion to determine all influencing nodes. Second, conditions are propagated within the computed backward slice. Then, the incoming condition of the CIA criterion is taken and a forward slice is computed

5.3 Approach

The CA-CIA approach determines the configuration-aware change impact by starting at the presence conditions in the CSDG and propagating resulting impact conditions as well as reaching conditions to CSDG nodes. A naïve approach would be to start at the presence conditions from the CSDG and globally propagate all conditions until every CSDG node is annotated with the appropriate impact condition and then performing a forward slice starting at the statements of interest. However, it is expected that most changes will affect only a subset of the code. Propagating the conditions globally and in advance would thus be too costly. Therefore, our CA-CIA algorithm propagates conditions only in the required domain.

Figure 8 shows the steps for computing the configuration-aware change impact. The approach starts at the node for which we want to compute the change impact, i.e, the *CIA criterion*. First, it computes the backward slice for the CIA criterion by computing all statements that possibly have an influence on the CIA criterion. This set of nodes is the so-called *domain*. It then takes presence conditions from the CSDG and propagates them together with the reaching conditions according to propagation rules within the computed backward slice, such that impact conditions and reaching conditions are determined for all incoming edges of the CIA criterion. Next, it determines the actual change impact by performing a forward slice, starting with the CIA criterion and propagating the impact conditions and reaching conditions.

Further, the approach needs to know the impact conditions and reaching conditions for all nodes in the change impact to compute the correct conditions in the forward slice. Figure 9 illustrates the situation where some nodes visited during the forward slice do not have impact conditions on the incoming control flow edge. The box with the dashed border denotes the initial domain, i.e., the set of nodes and edges that already have been considered in the initial backward slice. The highlighted control edge from the filled node is not part of the domain and does not yet have conditions. However, a condition on this edge might also have an influence on the conditions of the impacted statements. Therefore, the node with this incoming edge is registered, and in the end, a backward slice and the propagation within this slice are performed in a second pass. We thus mark all nodes with incoming edges that were not in the initial domain. After collecting these nodes, we compute a backward slice to determine the final propagation domain, which contains all nodes that can possibly reach and influence the nodes in

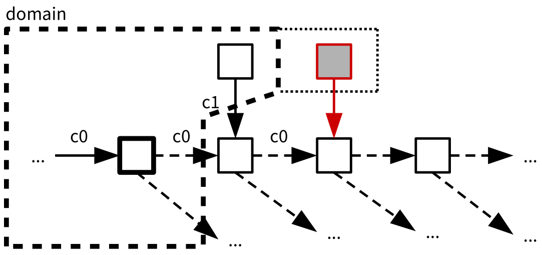


Fig. 9 Including further nodes in the propagation domain during the forward slice: The node with the bold border is the change impact criterion. The area surrounded by the dotted line depict the initial propagation domain. During the forward slice, the red node represents a node without a condition and therefore has to be included in the propagation domain in a second backward slice

the change impact. Finally, we apply the propagation algorithm in this domain and label all edges in the change impact with the correct impact conditions.

5.4 Algorithm

This section presents the algorithm for performing the CA-CIA as illustrated above based on the formal definitions of the CSDG and of the IC and RC functions. The inputs to the algorithm are the CSDG and the change impact criterion $criterion$. The algorithm iteratively computes updates of the impact conditions IC and reaching conditions RC. The result of the algorithm is the final impact condition function IC representing conditions for the change impacts on possibly influenced nodes. Specifically, algorithm 1 works in several phases:

- First, a backward slice starting at the CIA criterion is computed to determine the initial domain, i.e., the subgraph of the CSDG, which has influence on the criterion.
- Second, the variability information, i.e, the impact conditions and reaching condition sets, are propagated within this domain, i.e., impact conditions IC and reaching conditions RC are computed within the domain. In this way, the influencing variability information is brought to the CIA criterion. The presence conditions PC from the CSDG serve as initial values for the impact conditions IC. The reaching condition sets for all variables are empty initially.
- Next, a forward slice is performed that carries the impact conditions and reaching conditions from the CIA criterion along the dependency edges.
- Then, the nodes that have been visited during the forward slice and are not in the initial domain, are the starting point for a second backward slice.
- Once this second backward slice is computed resulting in an extended domain, another propagation phase starting at the CI criterion is performed to compute the final change impact conditions.

Algorithms 2–6 provide details for these individual steps.

Algorithm 1 CA-CIA algorithm.

```
1: procedure ConfigurationAwareCIA($CSDG = (V, E, T, C, PC), criterion$)
2:     $domain \leftarrow$ BackwardSlice($CSDG, criterion$)
3:     $(IC, RC) \leftarrow (PC, RC_0)$ with $RC_0(e)(v) = \emptyset, \forall e \in E, \forall v \in Vars$
4:     $(IC, RC) \leftarrow$ Propagate($CSDG, domain, (IC, RC)$)
5:     $((IC, RC), visited) \leftarrow$ ChangeImpact($CSDG, criterion, (IC, RC)$)
6:     $extendedDomain \leftarrow$ BackwardSlice($CSDG, visited \backslash domain$)
7:     $(IC, RC) \leftarrow$ Propagate($CSDG, extendedDomain, (IC, RC)$)
8:     return $IC$
9: end procedure
```

Algorithm 2 The backward slicing algorithm.

```
procedure BackwardSlice($CSDG = (V, E, T, C, PC), criterion$)
    $visited \leftarrow \emptyset$
    $queue \leftarrow criterion$
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        $visited \leftarrow visited \cup \{n\}$
        for $(p \rightarrow n) \in E$ do
            if $p \notin visited$ then
                $queue \leftarrow queue \cup \{p\}$
            end if
        end for
    end while
    return $visited$
end procedure
```

Algorithm 2 defines the backward slicing algorithm as presented by Ottenstein and Ottenstein (1984). Since the CSDG is an extended SDG, we can simply perform a graph traversal. Therefore, we first insert the CIA criterion into the queue and then determine all reachable nodes by following the edges backwards.

Algorithm 3 performs the propagation of impact conditions and reaching conditions in the domain, i.e., the forward propagation along the edges in the domain to carry the impact conditions and reaching conditions to the CIA criterion. It actually is a fixpoint algorithm, which propagates impact conditions and reaching conditions as described by the propagation cases in Sect. 5.5. The input parameters are the CSDG and the domain as well as the initial impact conditions and reaching conditions (empty initially). The algorithm uses a working queue containing nodes to consider. It starts by looking for nodes which have outgoing control edges with presence conditions by iterating over the nodes in the domain. Those nodes are put into the working queue to start propagation. Then, propagation of impact conditions and reaching conditions is done as long as the working queue is not empty. Thus, for each node in the queue, it will propagate the conditions along the outgoing edges to the next nodes. As soon as there is a change of impact conditions or reaching condition on an edge, the next node is put into the working queue. The propagation cases described in Sect. 5.5 guarantee termination of the algorithm. First, for the initial presence conditions a conjunction is built with the incoming conditions, cf. Fig. 10a. However, there is only a finite set of

Algorithm 3 The propagation algorithm distributing variability information within a domain.

```
procedure PROPAGATE($CSDG = (V, E, T, C, PC), domain, (IC, RC)$)
    $queue \leftarrow \emptyset$
    for nodes $n \in domain$ do
        if $\exists(n \rightarrow s) : PC(n \rightarrow s) \neq True$ then
            $queue \leftarrow queue \cup \{n\}$
        end if
    end for
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        for outgoing edges $e = (n \rightarrow s) \in domain$ do
            $((IC, RC), changed) \leftarrow$ MoveCondition($CSDG, e, (IC, RC)$)
            if $changed$ then
                $queue \leftarrow queue \cup \{s\}$
            end if
        end for
    end while
    return $(IC, RC)$
end procedure
```

edges carrying presence conditions and propagation is effective only the first time, i.e., combining it conjunctively a second time does not change the condition. Then only disjunctions are built when combining impact conditions, cf. Fig. 10b. This guarantees that the impact conditions will always get more general until possibly becoming true. Moreover, there is also only a finite set of assignments of conditions to variables and thus also the reaching conditions will stabilize.

Algorithm 4 is the change impact algorithm and performs a forward slice starting at the CIA criterion. The input parameters of the algorithms are the CSDG, the CIA criterion, and the impact conditions and reaching conditions as computed so far. The algorithm moves the propagated conditions and reaching condition sets during traversal and also collects the visited nodes.

Algorithm 4 The forward propagation algorithm for computing the change impact conditions starting at the CI criterion.

```
procedure CHANGEIMPACT($CSDG = (V, E, T, C, PC), criterion, (IC, RC)$)
    $visited \leftarrow \emptyset$
    $queue \leftarrow \{criterion\}$
    while $queue \neq \emptyset$ do
        $n \leftarrow$ removeFirst($queue$)
        $visited \leftarrow visited \cup \{n\}$
        for outgoing edges $e = (n \rightarrow s) \in E$ do
            ((IC, RC), _) $\leftarrow$ MoveCondition($CSDG, e, (IC, RC)$)
            if $s \notin visited$ then
                $queue \leftarrow queue \cup \{s\}$
            end if
        end for
    end while
    return $((IC, RC), visited)$
end procedure
```

Algorithm 5 is responsible for performing a single propagation step for a single edge. Its inputs are the CSDG, the edge and the current impact conditions and reaching condition sets. It returns possibly updated impact conditions and reaching condition sets and a flag indicating a change. The main work is done by algorithm *Propagate-Condition* which combines the conditions from the predecessor edges and the presence conditions of the edge itself. This is shown in Sect. 5.5 (Algorithm 6).

Algorithm 5 Computes the impact condition and reaching conditions for an edge by combining the conditions from the predessor edges and its presence condition.

```
procedure MoveCondition($CSDG = (V, E, T, C, PC), e, (IC, RC)$)
    $(ic_e, rc_e) \leftarrow$ PropagateConditions($CSDG, e, (IC, RC)$)
    $changed \leftarrow IC(e) \neq ic_e \vee RC(e) \neq rc_e$
    $(IC', RC') \leftarrow$ update($(IC, RC), (ic_e, rc_e)$)
    return $((IC', RC'), changed)$
end procedure
```

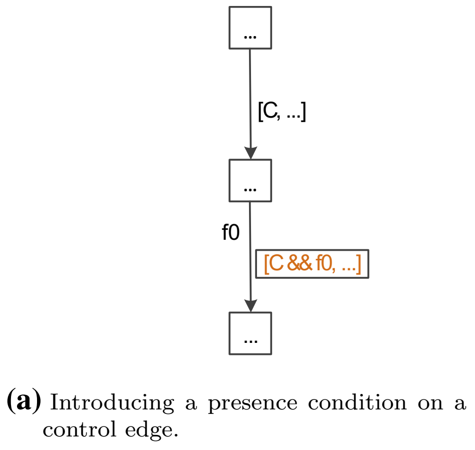
5.5 Propagation cases

This section describes the basic propagation cases as performed by Algorithm 6. Each case is illustrated by a fraction of the CSDG with three node levels. The upper node level are the predecessors, the middle node level contains the current node with the outgoing edge to process, and the lower level contains the successor node. The node types are not specified explicitly since they result from the content of the node and the edge types are specified by the line type. Solid lines represent control dependence edges and dashed lines represent data dependence edges. Furthermore, the impact conditions (IC) and reaching conditions (RC) are depicted by a tuple `[IC, RC]`. If the tuple contains three dots (…), the element at this position does not matter. The presence conditions (PC) on edges are shown without square brackets if present.

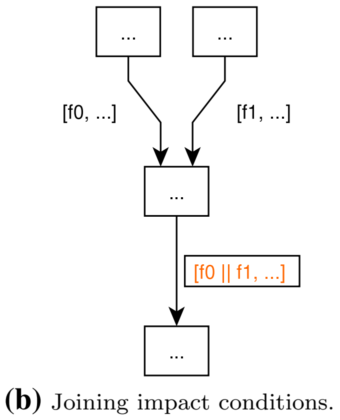
Case 1. Introducing a presence condition from a control edge In this case shown in Fig. 10a, the edge has a presence condition `f0`. Moreover, the incoming edge to the source node already contains some impact condition `C`. When propagating this incoming impact condition, it has to be combined with the presence condition `f0`. The outgoing edge thus gets an impact condition `C && f0`.

Case 2. Joining impact conditions In this propagation case depicted in Fig. 10b, the source node has multiple incoming control dependence edges, each carrying distinct impact conditions. The conditions of the incoming edges are combined disjunctively and the combined condition is then put on the outgoing edge. That means that the edge is enabled if either of the two conditions is fulfilled.

Case 3. Introducing and using new configuration variables In this case shown in Fig. 11a, a new configuration variable `x` is introduced in a predecessor node, which gets assigned a combination of two presence conditions `f0` and `f1`. This is forwarded in a reaching condition definition for variable `x` along a data dependence edge. Then this variable is used in an if statement. This results in the reaching condition being used in the impact condition `!(f0 && f1)` of the outgoing control dependence edge.

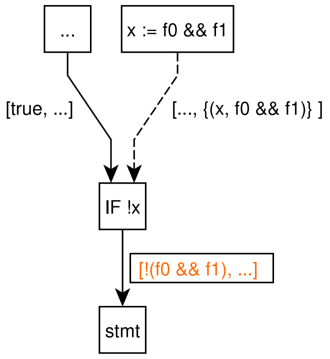


(a) Introducing a presence condition on a control edge.

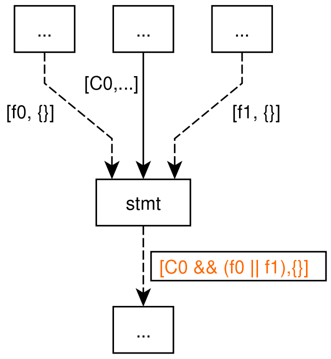


(b) Joining impact conditions.

Fig. 10 Basic cases of propagating impact conditions



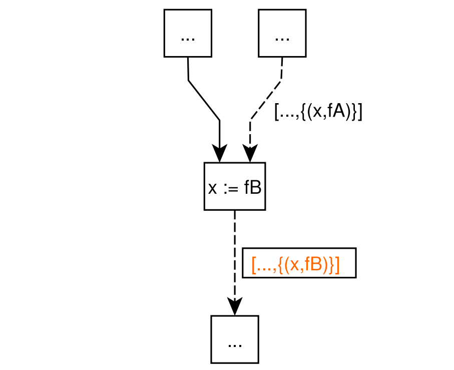
(a) Introducing and using new configuration variables. The reaching condition $(x, f0 \wedge f1)$ is used by branch statement `IF !x`. Therefore, a new impact condition is created by negating the reaching condition.



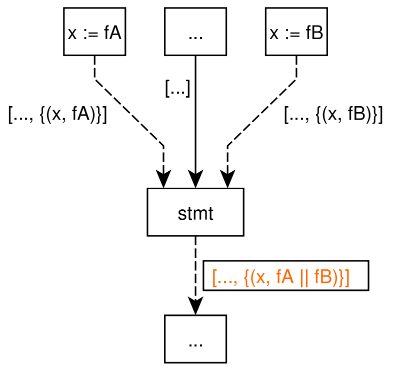
(b) Propagating impact conditions to data dependence edges. The outgoing edge will only be enabled if the statement is executed, i.e., $C0$ is satisfied, and if at least one incoming data edge is enabled, i.e., $f0 \vee f1$ is fulfilled.

Fig. 11 Propagation cases with interaction between impact and reaching conditions

Case 4. Propagating impact conditions to data dependence edges This case defines how conditions on incoming control and data dependence edges are propagated to an outgoing data dependence edge (cf. Fig. 11b). First, the impact conditions of incoming control dependence edges propagates to all outgoing data dependence edges, as the data dependence of the node only occurs if the node is executed. Therefore, the outgoing data dependence edge gets an impact condition `C0` in the example. Secondly, there are incoming data dependence edges with conditions `f0` and `f1`. However, the statement can only be executed if one of these edges provides data. The outgoing data dependence



(a) Updating a reaching condition set by a new reaching condition introduced by assigning configuration variable x.



(b) Joining reaching condition sets. The two incoming definitions of configuration variable x are combined disjunctively.

Fig. 12 Basic cases of propagating reaching conditions

edge is thus only enabled if the statement is executed and if there is incoming data, i.e., the impact condition `(C0 && (f0 || f1)` is satisfied.

Case 5. Updating reaching condition sets This case presented in Fig. 12a deals with propagating a reaching condition set across a node introducing a new configuration condition overwriting the existing one. As can be seen in Fig. 12a, the incoming data edge carries a reaching condition `fA` for variable `x`. The node redefines variable `x` with a new condition `fB`. Thus, the definition for `x` is overwritten and the outgoing data edge carries the definition `(x, fB)`.

Case 6. Joining reaching condition sets In this case a node has two incoming and one outgoing data dependence edge as can be seen in Fig. 12b. The reaching conditions are combined using a union operation but reaching conditions for the same variables `(x, fA)` and `(x, fB)` are combined using a logical OR operator.

Algorithm 6 describes how the impact conditions and reaching condition sets are computed. It implements the propagation cases from above. The input parameters are the CSDG, the edge, the current impact conditions and reaching conditions. The variable icc_e is used for the new impact condition for edge e if e is a control dependence edge. The variable icd_e is used for the new impact condition for edge e if e is a data dependence edge. The variable rcd_e is the reaching conditions set from data dependence edge predecessors.

In the first loop, the algorithm iterates over all predecessor edges and depending on the predecessor edge's type, the impact conditions and the reaching conditions are combined as illustrated by Case 2, 4 and 6. For combining the reaching conditions function, the operator $\oplus$ is used. It forms the union of two given reaching condition functions $RC(e_0)$ and $RC(e_1)$, but definitions affecting the same variables are joined by combining the condition disjunctively. For example, $\{(x, fA)\} \oplus \{(x, fB)\} = \{(x, fA \vee fB)\}$ as illustrated in Case 6.

After the for statement, the algorithm tests if the source node n contains an assignment statement and the receiver of the assignment is a variable contained in the reaching conditions. If so, the reaching conditions are updated such that (x, fB) is the only element using x on the left side in $RC(e)$ as (cf. Case 5). The else branch covers Case 3. It is tested if the node is an if statement and the expression contains a configuration variable. If so, the expression $expr$ is converted to an condition by replacing all occurrences of variables available in $RC(e)$. The resulting presence condition is then conjunctively combined with icc_e.

In the end of the algorithm, the new impact condition and reaching conditions are combined and returned. If the edge e is a control edge, the icc_e is conjunctively combined with the presence condition (cf. Case 1). If the edge e is a data dependence edge, the impact condition icc_e is conjunctively combined with icd_e (cf. Case 4) and the presence condition (cf. Case 1). The resulting impact condition and reaching conditions for edge e are returned.

Algorithm 6 Computes the condition for a provided edge.

```
procedure PropagateConditions($CSDG = (V, E, T, C, PC, RC), e = (n, s), (IC, RC)$)
    $icc_e \leftarrow \emptyset$                                ▷ Impact condition for control dependence edges
    $icd_e \leftarrow \emptyset$                                ▷ Impact condition for data dependence edges
    $rcd_e \leftarrow \emptyset$                                ▷ Reaching conditions for data dependence edges
    for $pe = (p \to n) \in E$ do
        if $T(pe) = ctrl$ then
            $icc_e \leftarrow icc_e \cup IC(pe)$                ▷ Case 2
        else if $T(pe) = data$ then
            $icd_e \leftarrow icd_e \cup IC(pe)$                ▷ Case 4
            $rcd_e \leftarrow rcd_e \oplus RC(pe)$              ▷ Case 6
        end if
    end for
    if $n =$ "x := fB" $\wedge x \in Vars$ then                 ▷ Case 5
        Update $rcd_e$ by inserting tuple $(x, fB)$.
        This possibly replaces any tuple with $(x, fB)$ on the left side.
    else if $n =$ "IF x" $\wedge \exists var \in expr : var \in Vars(rcd_e)$ then    ▷ Case 3
        Extract reaching condition $C = rcd(x)$ from expression
        $icc_e \leftarrow icc_e \cap C$ .
    end if
    if $T(e) = ctrl$ then
        return $(icc_e \cap PC(e), \emptyset)$                  ▷ Case 1
    else if $T(pe) = data$ then
        return $(icc_e \cap icd_e \cap PC(e), rcd_e)$           ▷ Cases 4, 1
    end if
end procedure
```

6 Evaluation

Our evaluation investigates the benefits regarding complexity reduction and the performance of the configuration-aware CIA approach in two use cases: (i) development and maintenance in domain engineering, i.e., for determining the different product variants affected by a change; and (ii) development and maintenance in application engineering, i.e., for determining code affected by a change made to a specific product variant. Specifically, we use product families of our industry partner to explore the following research questions:

RQ1 *How beneficial is the configuration-aware CIA for maintaining a product line?* Our approach is highly beneficial in situations of high variability complexity. We thus estimate the benefit by computing the degree and complexity of variability information a domain engineer needs to consider when determining the impact on product variants after changing code in a product line.

RQ2 *How beneficial is the configuration-aware CIA for maintaining a specific product variant?* This situation is common in clone-and-own product lines. Typically, in such product lines systems are configured in a two-stage derivation process, where some variability is resolved in first stage, e.g., by setting configuration options. Then in a second stage, developers adapt and fine-tune the product variant to meet specific customer needs. We estimate the benefit by computing the increased precision of the change impact, as a smaller change impact will reduce the development effort.

RQ3 *Is the performance of the configuration-aware CIA sufficient for realistic maintenance tasks?* A major problem of static program analysis is high run time complexity. We perform benchmarking to show the practicality and suitability of the approach in realistic maintenance tasks.

6.1 Case study and code base selected for the evaluation

To investigate these research questions, we applied our tool to real-world software systems provided by our industry partner Keba AG (http://www.keba.com), a medium-sized company developing and producing tools, hardware, and software for the industrial automation domain. One of their systems is KePlast (Lettner et al. 2013), a comprehensive solution for the automation of injection molding machines. The core of KePlast is a configurable control software framework which is implemented in a proprietary dialect of the IEC 61131-3 standard (IEC 2013), a widely-adopted programming language standard for implementing industrial automation systems. KePlast exists in multiple different product families addressing different market segments. Keba uses a custom-developed product configuration tool to select components from their KePlast platform based on customer requirements. The selected components are then adapted and extended by application engineers to meet specific customer needs not yet covered by the platform. In this stage, the derived software is still configurable by using load-time configuration options, the variability mechanism targeted by our approach. Two load-time variability mechanisms are used in the KePlast system: (i) There is a test predicate IS_LINKED to determine if certain modules have been linked in the first stage and are thus present in the system. Based on this test other program parts are then enabled or disabled at run time. (ii) There is a set of configuration variables loaded from configuration files at load time. The configuration-aware analysis relies on this information.

As a baseline for our evaluation we derived product variants from KePlast's families, which contain a maximum number of features selectable together in the product

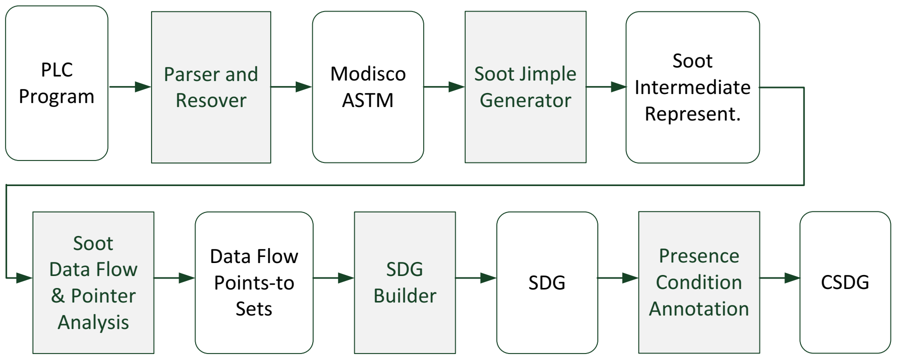


Fig. 13 Components of the program analysis tool chain: Parser and Resolver, Jimple Generator, and System Dependence Graph Builder

configuration tool. These maximum products do not contain the full code base of the product line, but are a good approximation for the purpose of our study. In particular, we used 9 different still configurable product variants with a size ranging from 53 kLOC (*family4*) to 302 kLOC (*family2*).

Furthermore, these maximum product variants of a family indeed have commonalities and differences. In the case of our industry partner's product families, the common code, i.e. the mandatory features, is a substantial part of the source code because it includes implementations for basic data structures and algorithms used in all product variants. We therefore concentrate on the variable part of the analyzed product families to obtain data that is most relevant for our research questions. Specifically, we only considered change impacts containing at least one impact condition. This was the case for 27% of all change impacts in our case study (median across product families).

6.2 Tool implementation and adaptation

We have developed a tool chain for implementing the CA-CIA approach, which is shown in Fig. 13. KePlast has been developed in a proprietary dialect of the IEC 61131-3 standard, for which no parser or compiler suitable for program analysis was available. We thus first had to create a parser frontend using the parser generator CoCo/R (Wöß et al. 2003) and an abstract syntax tree (AST) to represent the parsed source code. Our AST implementation is based on Modisco (Bruneliere et al. 2014), an implementation of the OMG ASTM standard that defines an AST meta model. For data flow analysis and pointer analysis we use Soot (Lam et al. 2011), an analysis framework originally developed for Java code analysis. Based on the AST, we thus generate input for the Soot framework which preserves the control flow and data semantics of the original program. Finally, the System Graph Builder creates the SDG which is then further annotated with impact conditions. A detailed description of the tool chain is available from Grimmer et al. (2016).

The CA-CIA method takes the CSDG and a CIA criterion as input. The result is a subgraph of the CSDG with the propagated impact conditions. The SDG browser based on the *Eclipse* framework allows visualizing the results for developers (cf. Sect. 4.4).

6.3 RQ1: Domain engineering

As argued above, the benefit of the configuration-aware CIA depends on the complexity of the variability information and the contradictions of the impact conditions, which allow pruning the change impact. Recall that in our approach variability information is propagated based on the program dependencies and is therefore usually not visible directly in the source code.

We define three *metrics* to characterize the variability of a code base.

The *Variability Complexity* is the ratio of edges with variability information to the total number of edges within a change impact. Our approach is more beneficial if more edges are annotated with variability information, as this would make a manual CIA even harder.

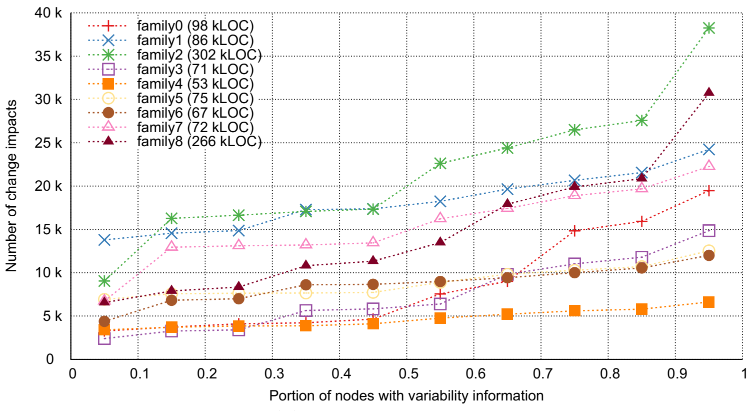
The *Variability Interaction Order* measures the average number of involved distinct configuration options in the variability information. This is computed by counting the number of distinct configuration options in a single impact condition and computing the mean of these numbers over all edges. For example, if the two conditions $a \wedge b$ and $\neg a \vee c$ are in a change impact, the interaction measure would be $\frac{2+2}{2} = 2$. This metric thus represents the interaction between different configuration options. The benefit of the approach increases with the order of the variability interaction, as manually analyzing complex interactions is hard to infeasible.

Contradicting Conditions Ratio The propagation of impact conditions may result in contradicting impact conditions, which can never become true, regardless of the product configuration. Such invalid edges allow removing statements from the change impact. Our measure is the ratio of invalid edges to the total number of edges. A higher value is better because the change impact is smaller and more precise.

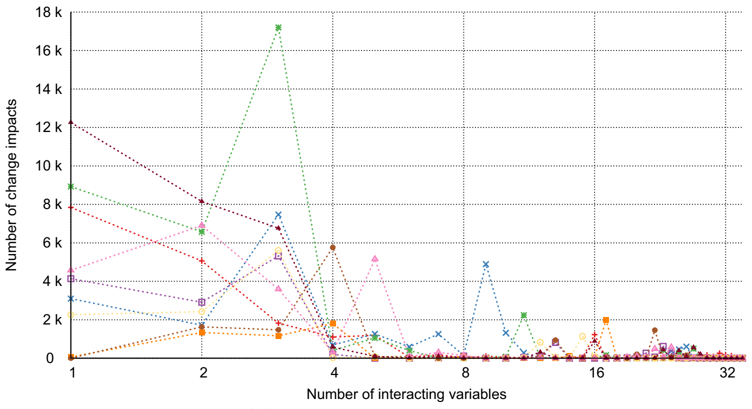
Method We performed the configuration-aware CIA for every single statement in the selected product families. This was done in two phases: in phase 1 we built the CSDG for the product family. This included parsing the source code, performing control flow, data flow and pointer analysis to build the SDG, and extracting the initial conditions from the source code. In phase 2 we performed the configuration-aware change impact analysis as outlined in Sect. 5. Specifically, we iterated over *all* statement nodes in the CSDG and performed a configuration-aware CIA.

Results Figure 14a shows the results of our evaluation for the metric *Variability Complexity*. The x-axis shows the complexity values, i.e, the portion of nodes with variability information. We grouped the values into intervals of width 0.1. Thus the x-axis represents intervals]0.0, 0.1[, [0.1, 0.2[, …, [0.9, 1.0]—an interval is denoted by its middle value on the x-axis. The y-axis is the number of change impacts contained in a certain interval. The chart shows the cumulative distribution of change impacts for these groups. For example, the data point at 0.65 of the *family2* says that approximately 25,000 change impacts had a variability complexity value in interval]0.0, 0.7[. Therefore, a impact condition was available for up to 70% of the statements in the change impact.

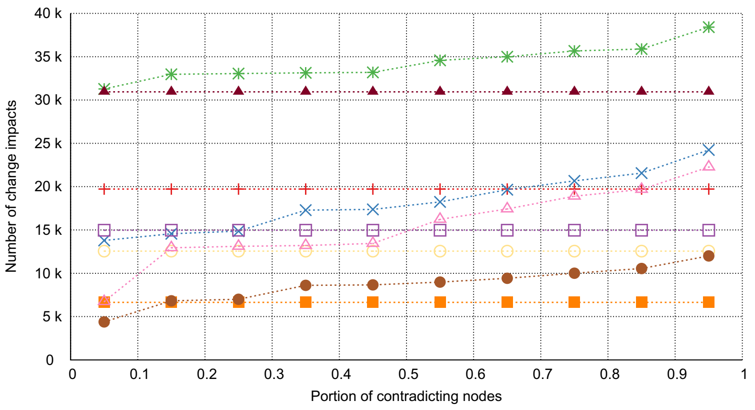
Figure 14b shows the results for metric *Variability Interaction Order*. The x-axis shows the average number of configuration options involved in the impact conditions of a change impact. The y-axis is the number of change impacts that have a certain



(a) RQ1–Variability Complexity.



(b) RQ1–Variability Interaction Order.



(c) RQ1–Contradicting Conditions Ratio.

Fig. 14 Results for RQ1 computed by applying our tool to a set of real-world product families provided by our industry partner

average number of involved configuration options. For example, the *family6* has a peak at $x = 4$, i.e., around 6000 change impacts involve on average 4 configuration options.

Figure 14c shows the results for metric *Contradicting Conditions Ratio*. The x-axis shows the portion of nodes that are invalid because of a contradicting condition. We again grouped the values into intervals of width 0.1. The x-axis therefore represents intervals]0.0, 0.1[, [0.1, 0.2[, …, [0.9, 1.0]—an interval is denoted by its middle value on the x-axis. The y-axis again represents the number of change impacts as in Fig. 14a. The chart shows the cumulative distribution of change impacts for these groups. For example, in *family2* at $x = 0.15$ the size of 33,000 change impacts could be reduced by up to 20%.

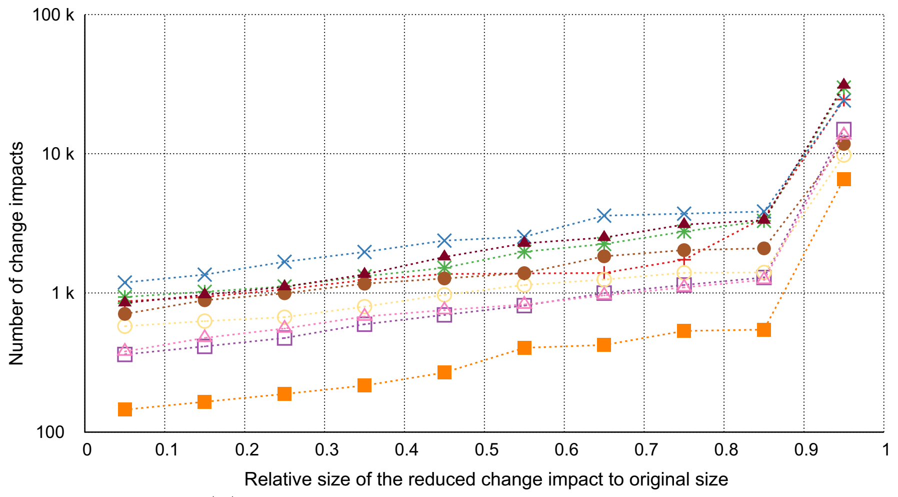
Discussion The results of metric *Variability Complexity* shown in Fig. 14a show that the approach is beneficial given the complexity of variability in real-world systems. The graph shows a cumulative distribution function. The steeper a line the more variability information is available in the computed change impacts. The chart also shows that larger product families provide even more variability information. For example, consider *family2*, the largest product family in our study. It first grows moderately until 0.45 but then it starts to grow faster, i.e., 50% of the statements in most change impacts had variability information available. When combining the data in Fig. 14a to one value, we see that overall the change impacts have 50–60% impact conditions available on average (median).

The results for metric *Variability Interaction Order* shown in Fig. 14b show a similar picture. Dealing with interactions involving 2 or 3 configuration options is already quite cumbersome. The results show that many change impacts had conditions with 5 and more configuration options. Other empirical studies have shown an interaction degree of 2 or 3 to be common (cf. Apel et al. 2013), similar to our results. However, we also found quite a few change impacts with higher numbers of 10 to 35 configuration options per impact condition, an order that is almost infeasible to comprehend by developers.

In Sect. 6.1 we described that our analysis is performed on product families which have already been partially configured and can be compiled. Therefore, the product families' source code does not contain any mutually exclusive feature implementations. So regarding the *Contradicting Conditions Ratio* we did not expect a significant increase of the precision of change impacts by finding contradicting impact conditions. This did in fact happen for the analyzed product families *family0*, *family4*, and *family5*. We see no increase in the corresponding lines in Fig. 14c, i.e., we could not remove any statements from the change impacts. However, we still could increase the precision of change impacts in the other product families. For example, the difference between the last two data points of *family2* shows that it was even possible to find approximately 2000 change impacts whose size could be reduced by 90–100%.

6.4 RQ2: Application engineering

For this use case we perform CIA for a specific product configuration, i.e., we remove elements from the result that are not reachable when evaluating the impact condtions



(a) RQ2. Relative Change Impact Size.

Fig. 15 Results for RQ2 computed by applying our tool to a set of real-world product families provided by our industry partner

in this specific configuration. Again, we distinguish between change impacts with and without variability information, because the latter one covers most commonly just library code, which is less interesting in terms of variability. We define metric *Relative Change Impact Size* to measure the reduced numbers of edges in the change impact after evaluating the impact conditions compared to the original change impact. The original change impact is the configuration-aware change impact but ignoring the impact conditions.

Method Analogously to the method for RQ1, we performed the configuration-aware CIA for every single statement of our product families (phase 1). However, for answering RQ2, we created concrete product configurations by randomly generating Boolean values for each known configuration option (phase 2). When computing the change impact, the impact conditions were then evaluated using these randomly generated values for the configuration options. We measured the reduction of the change impact size compared to its size not considering configurations. This was repeated 10 times with different generated values to compute an average for the *Relative Change Impact Size*.

Results Figure 15a shows the results for RQ2. Each line corresponds to one of the analyzed product families. The x-axis shows the relative size of the reduced change impact compared to the change impact without considering configurations. The y-axis represents the number of statements where the change impact could be reduced by that ratio. The results are again grouped into intervals of width 0.1. For example, the data point at $x = 0.85$ of *family4* means that the size of approximately 800 change impacts has been reduced by 10–20%.

Discussion We observe that evaluating the impact conditions after computing the change impacts reduces the size to 90% in most cases. Figure 15a shows a sharp edge at the end of all lines, which means most change impacts are in the last group.

There are also many change impacts that could be reduced to less than 80% of their size if not considering configurations. While we expected these numbers to be more distributed across the other groups, the results show that configuration does not equally impact all parts of a program. To answer RQ2, we conclude that evaluating the impact conditions yields noticeable benefit for maintenance because fewer statements have to be considered, although we expected to do better.

6.5 RQ3: Performance

One major problem of static program analysis techniques is commonly their analysis time. We therefore measured the analysis time of our tool when analyzing our industry partner's product families.

Method We separately analyzed the performance for building the CSDG and for performing the configuration-aware CIA. For analyzing the building of the CSDG we used 34 different product variants derived from the KePlast automation platform from our industry partner (cf. Grimmer et al. 2016 for details). We measured the time required to build the SDG and the peak memory consumption (Java heap space). We executed all performance runs 10 times and took the median of the results.

The performance of the change impact method were measured based on 9 different product variants of the KePlast platform with a size ranging from 53 kLOC to 302 kLOC. We measured the size of all configuration-aware change impacts for all statements and the time required to compute them (cf. phase 2 of RQ1 and RQ2).

The tool chain is written in Java. The experiments have been conducted using a Java 8 HotSpot 64-Bit Server VM on Windows 7 running on PCs with an Intel Core i7 with 2.93 GHz and 16 GB DDR3-SDRAM and 3.4 GHz and 16 GB DDR3-SDRAM, respectively.

Results The results of the performance evaluation for building the CSDG for 8 representative product variants (PV) of different size are shown in Table 1. The selected program sizes range from 53 kLOC to 302 kLOC. The table shows the different times for the different phases in the tool chain as well as the size of the SDG in kNodes. The total time for the whole analysis process is between 30 and 271 seconds. Number of nodes range from 106 to 708 kNodes. Figure 16 relates the size and the total analysis time for 34 product variants (the selected product variants contained in Table 1 have circles as markers).

The results of the evaluation of the CA-CIA are shown in Fig. 17. A cross in the chart represents a change impact. The x-axis lists the size of the change impacts in terms of included statements. The y-axis shows the time required to compute the change impacts in seconds.

Discussion The performance measuere obtained for building the CSDG showed a maximum time of 272 sec and a maximum number of nodes in the SDG of 708 kNodes, which is reasonable for a program size of 302 kLOC. Moreover, results show a strong linear correlation between program size and execution time (Pearson's correlation coefficient 0.982; p value $< 2.2 \times 10^{-16}$). From these results it can be deduced that the tool chain scales well to large-scale industrial applications.

Table 1 Results from the run-time performance evaluation

PV	Size (kLOC)	PM (GB)	AST (sec)	JCG (sec)	PDG (sec)	SDG (sec)	Total (sec)	SDG Size (k#Node)
1	53	1.5	2.8	2.5	11.2	14.2	30.7	106
2	60	2.1	3.9	3.2	20.1	22.3	49.5	164
3	81	1.8	3.9	4.0	22.4	19.5	49.8	187
4	98	3.3	4.9	5.4	20.9	42.8	74.0	188
5	205	7.3	10.2	16.5	107.6	36.8	171.1	416
6	253	8.0	12.2	23.5	130.5	49.9	216.1	604
7	265	9.4	12.7	24.4	135.4	53.8	226.3	626
8	302	9.5	14.9	29.8	163.9	62.9	271.5	708

The size of the product variants (*PV*) is specified in lines of code (*Size*). *PM* indicates the peak memory consumption. *AST* represents the time needed for parsing and building the AST. *JCG* is the time for the Jimple code generation. *PDG* specifies the time required to build the PDGs including their instantiation, *SDG* is the time required for building the SDG. *Total* is the overall time needed. *SDG Size* shows the size of the resulting SDGs in number of nodes

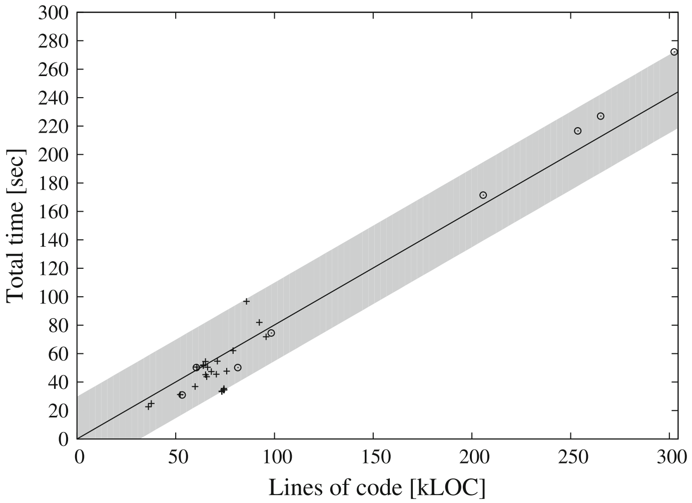


Fig. 16 Timings tested on 34 product variants

The results for the performance of the CIA approach showed that the computation of a change impact is fast and never takes longer than 3.5 seconds. It also seems that the time required to compute a change impact does not depend on its size because we cannot observe any linear or higher-order dependency on the change impact size. We assume the time is dominated by the first step of the CA-CIA algorithm, i.e., computing the backward slice to determine the possible influencing conditions. Moreover, computing the change impact is negligible compared to the time required to build the SDG.

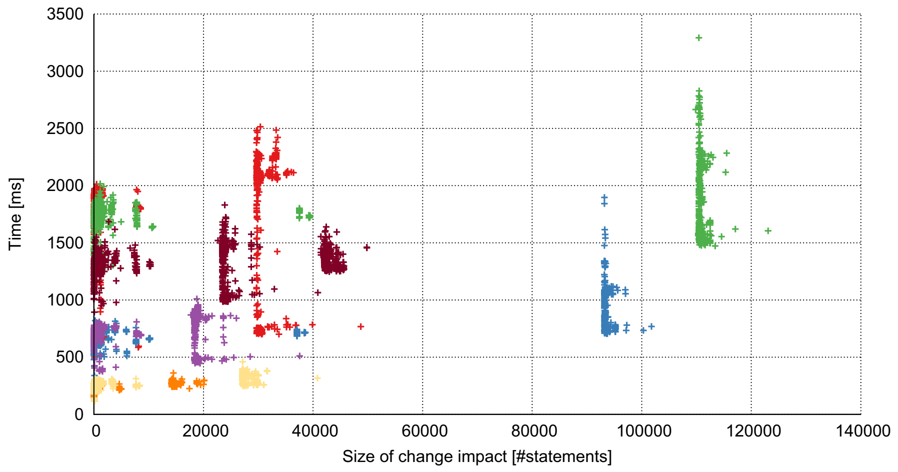


Fig. 17 RQ3—Average time required to compute configuration-aware change impacts of a specific size

6.6 Threats to validity

There is a potential bias caused by the selection of product families in a specific application domain that have been developed in a specific programming language. We thus present detailed evaluation results and avoid generalizations of how well the approach would work for other programming languages and PLs. However, our evaluation focuses on load-time configuration options, a variability mechanism that is widely used in all programming languages. Also, given that companies typically do not provide access to data about their product lines we believe that our evaluation results are valuable and promising.

Specifically, our evaluation is based on partially configured product families, that have been derived from a product line by selecting all features defined in a custom-developed configurator. We could not analyze the full code base of the product line, as the variability information is stored implicitly in the configurator. As a result the evaluation is based on a less configurable code base, e.g., certain alternative features are not included. However, this means that the results would be even more favorable for the full code base.

A prerequisite of our approach is that the mechanism for implementing variability in source code is known. This could be a problem in software projects without any conventions for this aspect, as extracting the initial conditions might not be possible. However, related work (Berger et al. 2010) indicates that most of the time it is known how variability is realized. Furthermore, Reisner et al. (2010) show that open source systems heavily use configuration options to implement variability.

The analyzed product families are executed in a runtime environment that provides library functions, whose implementation is unknown. This is also a source of imprecision when building the CSDG. However, such system functions occur in almost all execution environments and must be handled appropriately. Our implementation handles this problem by assuming the worst case if something is not known. For example, if a system variable returns a reference to a variable and this reference cannot be determined, we assumed that every reference may be returned. This is the common strategy to preserve soundness but sacrifices precision.

We have taken several measures to mitigate the risk of incorrect computations: we use the data flow and pointer analysis of the widely used Soot tool suite to build the SDG, so we have a high confidence that this part of the implementation produces correct results. Furthermore, the part of our implementation that extracts variability information from source code has already been reviewed by a developer of our industry partner in a qualitative process in our previous work (Angerer et al. 2014). Finally, we performed unit testing and manual reviews for the results based on test input. Furthermore, the tool suite was used on large-scale industrial systems and our industry partners selectively validated the results.

7 Related work

We structure our comparison to related work into work concerning (i) tracking configuration options, (ii) variability-aware program analysis, and (iii) program slicing and change impact analysis.

Tracking configuration options Lillack et al. (2014, 2017) developed an approach for tracking load-time configuration options. They use a modified taint analysis to determine source code depending on configuration options and also compute conditions for the presence of code elements. The propagation phase of our algorithm is comparable to their taint analysis, but their tool computes a different information. They create mappings between configuration options and source code statements whereas our approach globally propagates variability information and provides this information for an arbitrary change impact. Lillack et al. (2014) implemented their approach in a tool called *LOTRACK* by modifying Soot's taint analysis. The implementation of our approach works at a higher level, i.e. the CSDG, and is therefore less dependent on the analysis implementations required to build the CSDG. We think this is an important property because modifying existing implementations is often difficult and error prone.

Xu et al. (2013) introduce a tool named SPEX that analyzes a program's source code to infer constraints for configuration options by tracking the data-flow of each configuration variable. They also use program slicing to reduce the domain to consider but only on a configuration variable's data flow graph. The reduced domain is then used to infer constraints for configuration options by analyzing the statements using them. This is related to our approach, as we also analyze how a configuration variable is used, by considering the data flow graph of the variable. However, their two-phase algorithm may infer unsatisfactory constraints, whereas the fixpoint algorithm we use computes an over-approximation, i.e., the results are sound but may provide false positives.

Variability-aware program analysis Several authors have presented analysis techniques considering program-level variability mechanisms. Variability-aware program analysis techniques exploit the similarities among individual variants to reduce program analysis effort. Kästner et al. (2011) present the tool *TypeChef*, which parses unpreprocessed C source code and encodes the variability in the abstract syntax tree using presence conditions. Brabrand et al. (2012) present an approach to automatically lift standard intraprocedural dataflow analyses to feature-sensitive analyses. A lifted analysis can then analyze the whole program space of a preprocessor-based product line at once. Liebig et al. (2013) provide variability-aware type checking and a variability-aware liveness analysis for preprocessor-based product lines, also based on encoding variability using Boolean formulas. They also use a fixpoint algorithm to compute the result for single program elements, as in all dataflow analyses. The work introduces the patterns of *early joining* and *late splitting*, which is crucial for scalability. While we also use these patterns to keep the number of explicitly stored result data minimal, our approach differs with respect to when the variability information is added. The above approaches extract variability information from a variable AST, then build a variable control flow graph (CFG) and finally carry out the variability-aware analysis. In our case, the variability information is added in the last step after building the SDG. However, considering variability in the AST and CFG could further increase the precision of building the CSDG and therefore improve the overall results of our analysis. But since in our context variability is mainly implemented using run-time configuration options instead of `#IFDEF` directives, we do not need variability-aware parsing.

Our discussion shows the two main strategies proposed in the literature to make existing program analysis techniques variability-aware: (i) program analysis can be lifted by considering variability already in the parsing stage; or (ii) analysis can be delayed by considering and recovering variability only when needed. The delayed variability analysis works by performing an analysis completely variability-oblivious, i.e., ignoring any variability in the first place, and then recovering the variability and augmenting the results. The drawback of this strategy is the loss in precision because the delayed variability analysis needs to overapproximate situations to be sound. In Angerer et al. (2017) we provide an in-depth comparison of our approach with SPLLIFT (Brabrand et al. 2012), including an analysis and discussion of the trade-offs regarding precision and run-time performance. The results of our experiment show that the delayed strategy is significantly faster but at the same time typically less precise.

Kästner et al. (2014) provide comprehensive support for creating feature models, locating feature code based on manual condition definitions, and rewriting code into conditional compilation and feature modules. Our approach also employs program analysis techniques to build the CSDG. However, our goal is not to locate or rewrite feature code, although the propagation of the conditions could be used for feature location.

Zhang and Ernst (2013) present the ConfDiagnoser approach which uses static analysis, dynamic profiling, and statistical analysis to reveal the root cause of configuration errors. Our approach also exploits static analysis techniques and uses configuration files as input, however, we do not focus on determining configuration options leading to an error. Reisner et al. (2010) empirically analyze how configuration options affect program behavior. In particular, the authors use symbolic evaluation to discover how run-time configuration options affect line, basic block, edge, and condition coverage for different subject programs. We also compute the influence of configuration options to statements. Our approach is not limited to compute the impact when changing a configuration option, but works with arbitrary statements.

Program slicing and change impact analysis Our approach builds on earlier CIA research like (Arnold 1996), or program slicing approaches as summarized in Xu et al. (2005). In particular, the implementation of our approach is based on the interprocedural program slicing technique described by Horwitz et al. (1990). The approach presented in this paper further uses the CSDG we have presented in our previous work (Angerer et al. 2014). In contrast to most existing change impact analysis approaches, our approach extracts and propagates conditions over the CSDG, similar to Snelting (1996), who aims at increasing the accuracy of slices by eliminating impossible execution paths. This is done by extracting path conditions from all conditional statements and employing an SMT solver to eliminate infeasible paths. In contrast, we only extract conditions considering configuration variables and only deal with variability that can be represented by Boolean formulas. Since we extract fewer conditions and can use a SAT solver for reasoning, our approach can deal with much larger programs compared to data reported by Hammer et al. (2006).

8 Conclusions and future work

This paper presented a configuration-aware change impact analysis approach, which is based on program slicing techniques and a conditional system dependence graph (CSDG), an extension of an SDG. The approach can deal with load-time configuration options. It propagates conditions representing the variability of the analyzed software system. We implemented the approach based on the Soot analysis framework (Lam et al. 2011). We additionally built a front end for Soot for an industry partner's software systems and programming language. We then evaluated the benefit and performance of our approach on 9 product families of the industry partner to investigate the distribution and complexity of variability information and the required analysis time.

Regarding RQ1 we found high variability complexity demonstrating the benefit for our approach in industrial-size systems. Specifically, in more than 50% of all computed change impacts, 50–60% of the impacted nodes carried variability information. Regarding Variability Interaction Order, we also observed very high interaction orders of up to 35 configuration options. With respect to RQ2, the evaluation showed a noticeable benefit because fewer statements have to be considered in maintenance tasks. Regarding RQ3, the runtime performance of building the SDG and computing the change impacts allow the use of our technique in typical development and maintenance tasks.

As future work, we plan to formalize our approach using the framework by Midtgaard et al. (2014), which allows to derive variability-aware analyses for software product lines. We want to show that our configuration-aware CIA can be derived from a standard CIA approach. The evaluation of the approach as presented in this paper has been done based on industrial software systems implemented in an implementation of the IEC 61131-3 standard from our industry partner. However, the approach is also applicable to other imperative and object-oriented languages. For example, in Angerer et al. (2017) we have presented a study, which is based on Java product lines.

Furthermore, we have extended our tool to support multi-language software systems, e.g., with parts written in IEC 61131-3 as well as Java (Angerer 2014). This is important for industrial software systems, which often comprise subsystems written in different languages. For instance, it is common practice to load configuration options in one subsystem and communicate this information to other subsystems using different languages. Performing a configuration-aware CIA in such an environment, will require extensions to our approach allowing to jointly analyze the subsystems.

Acknowledgements Open access funding provided by Johannes Kepler University Linz. The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Keba AG is gratefully acknowledged. We thank Peter Feichtinger for implementing the SDG Browser tool.

References

Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, Burlington (2001)

Angerer, F.: Variability-aware change impact analysis of multi-language product lines (doctoral symposium paper). In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14). ACM, New York, NY, USA, pp. 903–906 (2014)

Angerer, F., Prähofer, H., Lettner, D., Grimmer, A., Grünbacher, P.: Identifying inactive code in product lines with configuration-aware system dependence graphs. In: Proceedings of the 18th International Software Product Line Conference, SPLC'14 (2014)

Angerer, F., Grimmer, A., Prähofer, H., Grünbacher, P.: Configuration-aware change impact analysis. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, Lincoln, Nebraska, USA, ASE'15, pp. 385–395 (2015)

Angerer, F., Grünbacher, P., Prähofer, H., Linsbauer, L.: An experiment comparing lifted and delayed variability-aware program analysis. In: 33rd IEEE International Conference on Software Maintenance and Evolution. IEEE, Shanghai, China (2017)

Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)

Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., Garvin, B.: Exploring feature interactions in the wild: the new feature-interaction challenge. In: Proceedings of the 5th International Workshop on Feature-Oriented Software Development, ACM, New York, NY, USA, FOSD'13, pp. 1–8 (2013)

Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)

Berger, T., She, S., Lotufo, R.: Variability Modeling in the Real: A Perspective from the Operating Systems Domain. Measurement, pp. 73–82 (2010)

Black, S.: Computing ripple effect for software maintenance. J. Softw. Maint. Evol. Res. Pract. **13**(4), 263–279 (2001)

Bohner, S.: Extending software change impact analysis into COTS components. In: Proceedings 27th Annual NASA Goddard/IEEE Software Engineering Workshop (2002)

Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P.: Intraprocedural dataflow analysis for software product lines. In: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, ACM, New York, NY, USA, AOSD'12, pp 13–24 (2012)

Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)

Chen, K.C.K., Rajich, V.: RIPPLES: tool for change in legacy software. In: Proceedings IEEE International Conference on Software Maintenance (2001)

Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, Reading (2001)

Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL'77, pp. 238–252 (1977)

Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)

Feichtinger, P.: Dependency Browser for Program Comprehension and Change Impact Analysis of PLC Programs. Master's thesis, Johannes Kepler University Linz, Austria (2017)

Ferrante, J., Ottenstein, K.J., Warren, D.J.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)

Grimmer, A., Angerer, F., Prähofer, H., Grünbacher, P.: Supporting program analysis for non-mainstream languages: experiences and lessons learned. In: Proceedings 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Osaka, Japan, SANER'16, pp. 460–469 (2016)

Hammer, C., Krinke, J., Snelting, G.: Information flow control for Java based on path conditions in dependence graphs. In: Proceedings IEEE International Symposium on Secure Software Engineering, pp. 87–96 (2006)

Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)

International Electrotechnical Commission (IEC) 61131-3:2013, Programmable controllers - Part 3: Programming languages, International Electrotechnical Commission, 2013

Jász, J., Beszédes, A., Gyimóthy, T., Rajlich, V.: Static execute after/before as a replacement of traditional software dependencies. In: IEEE International Conference on Software Maintenance, ICSM, pp. 137–146 (2008)

Kästner, C.: Virtual separation of concerns: toward preprocessors 2.0. Inf. Technol. **54**(1), 42–46 (2012)

Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proceedings of the 30th Int'l Conference on Software Engineering, ACM, ICSE'08, pp. 311–320 (2008)

Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. SIGPLAN Notes **46**(10), 805–824 (2011)

Kästner, C., Dreiling, A., Ostermann, K.: Variability mining: consistent semi-automatic detection of product-line features. IEEE Trans. Softw. Eng. **40**(1), 67–82 (2014)

Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice Hall, Englewood Cliffs (1988). https://doi.org/10.1002/spe.4380180707

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. ACM Comput. Surv. **28**, 220–242 (1997)

Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (2011)

Lettner, D., Petruzelka, M., Rabiser, R., Angerer, F., Prähofer, H., Grünbacher, P.: Custom-developed vs. model-based configuration tools: experiences from an industrial automation ecosystem. In: Proceedings of the 17th International Software Product Line Conference (SPLC'13 Workshops), ACM, New York, NY, USA, pp. 52–58 (2013)

Lettner, D., Angerer, F., Grünbacher, P., Prähofer, H.: Software Evolution in an Industrial Automation Ecosystem: An Exploratory Study. In: Proceedings International Euromicro Conference on Software Engineering and Advanced Applications, Verona, Italy (2014a)

Lettner, D., Angerer, F., Prähofer, H., Grünbacher, P.: A Case Study on Software Ecosystem Characteristics in Industrial Automation Software. In: Proceedings of the International Conference on Software and Systems Process, Nanjing, China, ICSSP'14 (2014b)

Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE'10, pp 105–114 (2010)

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C.: Scalable analysis of variable software. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, ESEC/FSE'13, pp 81–91 (2013)

Lillack, M., Kästner, C., Bodden, E.: Tracking load-time configuration options. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE'14, pp. 445–456 (2014)

Lillack, M., Kästner, C., Bodden, E.: Tracking load-time configuration options. IEEE Trans. Softw. Eng. (2017) https://doi.org/10.1109/TSE.2017.2756048

Linsbauer, L., Angerer, F., Grünbacher, P., Lettner, D., Prähofer, H., Lopez-Herrejon, R., Egyed, A.: Recovering feature-to-code mappings in mixed-variability software systems. In: Proceedings of the 30th International Conference on Software Maintenance and Evolution, ICSME'14 (2014)

Louridas, P.: Static code analysis. IEEE Softw. **23**(4), 58–61 (2006)

Midtgaard, J., Brabrand, C., Wasowski, A.: Systematic Derivation of Static Analyses for Software Product Lines. In: Proceedings of the 13th International Conference on Modularity, ACM, New York, NY, USA, MODULARITY'14, pp. 181–192 (2014)

Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, Burlington (1997)

Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer New York Inc, Secaucus (1999)

Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. SIGPLAN Notes **19**(5), 177–184 (1984)

Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, New York (2005)

Reisner, E., Song, C., Ma, K.K., Foster, J.S., Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, ACM, pp. 445–454 (2010)

Rubin, J., Chechik, M.: A Framework for managing cloned product variants. In: Proceedings of the 35th International Conference on Software Engineering, ICSE'13, pp. 1233–1236 (2013)

Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond. Springer, Berlin, Heidelberg, SPLC'10, pp. 77–91 (2010)

Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: Cousot, R., Schmidt, D. (eds.) Static Analysis SE-23, Lecture Notes in Computer Science, vol. 1145, pp. 332–348. Springer, Berlin Heidelberg (1996)

Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–754 (2005)

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. (CSUR) **47**, 1–45 (2014)

von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: from compile-time to load-time variability. J. Log. Algebraic Methods Program. **85**(1, Part 2), 125–145 (2016)

Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., Bodden, E.: Variational data structures: exploring tradeoffs in computing with variability. In: Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2014, pp. 213–226 (2014)

Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE'81, pp. 439–449 (1981)

Wöß, A., Löberbauer, M., Mössenböck, H.: LL(1) conflict resolution in a recursive descent compiler generator. In: Modular Programming Languages, Lecture Notes in Computer Science, vol. 2789. Springer Berlin Heidelberg, pp. 192–201 (2003)

Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)

Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., Pasupathy, S.: Do not blame users for misconfigurations. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, SOSP'13, pp. 244–259 (2013)

Zhang, S., Ernst, M.D.: Automated diagnosis of software configuration errors. In: 35th International conference on software engineering, San Francisco, CA, USA, pp. 312–321 (2013)