

Chapter 2

Software Families, Software Products Lines, and Dataflow Analyses

Abstract In this chapter we review essential concepts we explore in this work. Firstly, we review software families and software product lines, since the problem we address here is critical in these contexts. We show the basic concepts and then move towards conditional compilation with preprocessors, a widely used mechanism to implement features in industrial practice. Despite the widespread usage, conditional compilation has several drawbacks. We then present the Virtual Separation of Concerns (VSoC) approach, which can minimize some of these drawbacks. In this work, we intend to address the lack of feature modularity. Thus, we need to catch dependencies between features and inform developers about them. To do so, we rely on dataflow analyses, the last topic we review in this chapter.

Keywords Software families · Software product lines · Preprocessors · Virtual separation of concerns · Modularity · Dataflow analysis · Reaching definitions

2.1 Software Families and Software Product Lines

Software development often relies on version control systems like SVN [19] and CVS [8]. As we implement new functionalities, we put them under version control. In addition, we may remove or modify a functionality that for some reason did not work as expected. In this context, we have one software system per instant of time, as illustrated in Fig. 2.1. Notice that this system contains a base component (*circle*), which is common to all subsequent system versions.

However, having one system per instant is not convenient for large scale software production. For example, if a customer requires a system composed of the *circle*, *square*, and *triangle* components, we are not able to deliver such a product immediately because there is no version in the timeline that contains these three components—*circle*, *square*, and *triangle*—together. What we could do is to find a product that is an approximation of the one we need to build. Taking ± 3 as an example, we remove the *diamond* and then add the *triangle* component. However, removing and adding components may be a non-trivial task, being error-prone and increasing effort. In particular, *square* and *triangle* might have never worked together, which might increase our test case suite. As can be seen, all these activities may impact on time-to-market.

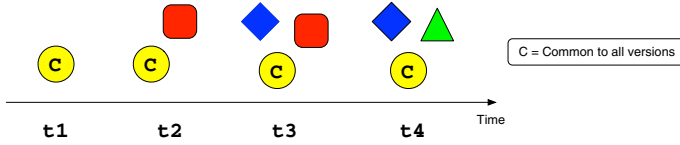


Fig. 2.1 One product per time

To improve this process, instead of having only one system per instant of time, we define a set of systems. This set contains similar systems differentiated by features [7]. We consider this set to constitute a *software family*, since we study the commonalities of the set and then the variabilities of the individual family members [21]. Although these members contain significant differences, it usually pays to learn the common properties of all systems before studying the details of any one [21]. This enables software reuse, for example, being important to deal with customers request more easily.

To enable the systematic construction of these individual systems with mass customization, we need to consider a product line. Basically, every software product line is a software family, but the opposite is not true. We explain product lines in what follows.

A Software Product Line (SPL) is *a set of software-intensive systems that share a common, managed set of features satisfying 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* [7]. In this context, core assets means any artifact used to instantiate more than one product [18]. Notice that these assets are not only code artifacts, but also any element used to develop the final product. This way, requirements, the software architecture, binary files, image files, test cases, sound files, and documentation are examples of potential core assets. Again, instead of having one product per instant of time, we have a set of core assets. Customers choose their particular product configurations, and then we build the product by reusing the core assets, usually bringing significant productivity and time-to-market improvements [7, 17, 22].

SPLs define two processes. The first one comprehends the core assets development. It establishes the reusable platform and consequently the commonalities and variabilities of the SPL, being known as *Domain Engineering*. The second one is responsible for deriving product line applications based on the predefined core assets. Therefore, it ensures the correct binding of the variabilities according to the application-specific needs. Such product development process is known as *Application Engineering*.

Some advantages of adopting the SPL approach are outlined below:

- **Reduction of development costs:** when artifacts are reusable in several different products, this implies a cost reduction for each product, since there is no need to develop such components from scratch. However, there is a cost associated with the beginning of the SPL design. In other words, before building final products, it is necessary firstly to design and implement the core assets so that we can reuse them. This way, before having one single product, we already have an upfront

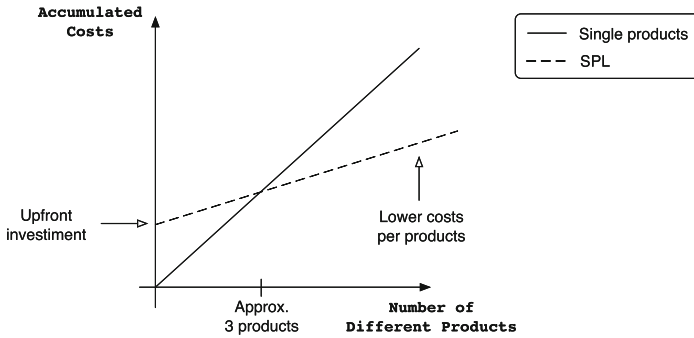


Fig. 2.2 Costs to develop single systems compared to the SPL engineering

investment. Empirical studies reveal that the upfront investments to design a SPL from scratch usually pay-off around three products [7]. Therefore, the accumulated costs are the same when developing three products without the SPL engineering and three products by using the SPL engineering (see Fig. 2.2). However, when the number of products increases, we have lower costs per product with the SPL approach;

- **Enhancement of quality:** the core assets of a SPL are reused in many products. In this way, they are tested and reviewed many times, which means that there is a higher chance of detecting faults and correcting them, which improves the quality of the products;
- **Reduction of time-to-market:** initially, the time-to-market of the SPL is high, because the core assets must be developed first. Afterwards, the time-to-market is reduced, because many previously developed components might be reused for each new product.

However, to gain all of these advantages, managing the SPL features in a suitable way is essential. In this context, features are the semantic units by which different products within a SPL can be differentiated and defined [27], playing a key role for mass customization. Features are also defined as prominent and stakeholder visible aspects, qualities, or characteristics of a software system or systems [10]. The Feature-Oriented Domain Analysis (FODA) approach is a domain analysis that focuses on the description of SPL commonalities and variabilities by means of features.

To illustrate feature examples, we use the *Best Lap* product line.¹ *Best Lap* is a race game where the player tries to achieve the best time in one lap and qualify for the pole position. It has approximately 15KLOC and is highly variant due to portability constraints: it should run on several platforms. In fact, the game is deployed on 65 devices [3]. Figure 2.3 illustrates the race game.

¹ *Best lap* is a commercial product developed by Meantime Mobile Creations. <http://www.meantime.com.br/>.

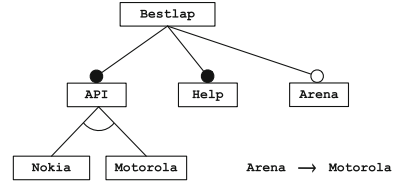


Fig. 2.3 Help (mandatory) and arena (optional) features in *Best Lap*

In this context, there are functionalities that are common to all *Best Lap* products. For instance, the menu item “Help” is mandatory, which means that all products contain this item. On the other hand, depending on the mobile device, some features are not available. For example, the *arena* feature (see the “Arena” menu item in Fig. 2.3) is an optional feature for publishing the scores obtained by the player on the network. This way, players around the world are able to compare their results. In addition, we may have features not visible for some stakeholders. For example, each device uses an API provided by the mobile manufacturer. Because it is not possible to use two APIs from two different manufacturers in the same product, we say that API is an alternative feature (or mutually exclusive feature).

The FODA approach represents in a compact way all possible products of a SPL by using a so called feature model. Besides describing features, the feature model provides a feature diagram (tree) and defines constraints between features. Figure 2.4 illustrates a small feature model of *Best Lap*. The feature diagram depicts a hierarchical decomposition of features with mandatory (*filled circle*), optional (*open circle*), and alternative (*open arc*) relationships. As can be seen, we have two feature constraints in this feature model. Firstly, *Nokia* and *Motorola* are alternative. Thus, they can never be together in the same product. Secondly, the *arena* feature only works with the *Motorola* API, so *Arena* \rightarrow *Motorola*.

Conceptually, a feature model is a propositional logic formula [4]. For instance, the above feature model is captured by the formula:

Fig. 2.4 Bestlap feature model**Fig. 2.5** Arena feature implemented with conditional compilation

```

public void computeLevel() {
    ...
    ...
    totalScore = ...;
    ...
    ...
    #ifdef ARENA
    NetworkFacade.setScore(totalScore);
    NetworkFacade.setLevel(this.getCurrentLevel());
    #endif
}

```

$$\psi_{\text{FM}} = \text{API} \wedge \text{Help} \wedge (\text{Nokia} \leftrightarrow \neg \text{Motorola}) \wedge (\text{Arena} \rightarrow \text{Motorola})$$

This feature model corresponds to the following set of valid configurations:

$$\llbracket \psi_{\text{FM}} \rrbracket = \{ \{ \text{API}, \text{Help}, \text{Nokia} \}, \\ \{ \text{API}, \text{Help}, \text{Motorola} \}, \\ \{ \text{API}, \text{Help}, \text{Motorola}, \text{Arena} \} \}$$

2.1.1 Conditional Compilation

Features are often implemented using mechanisms like preprocessors [2, 11, 15]. Conditional Compilation is a well-known and widely used mechanism for handling variabilities in languages such as C and C++. This mechanism is used at pre-compile time. The preprocessor analyzes the code that should be compiled or not based on directive tags. In this manner, if the directive tag is true, the code must be compiled. Otherwise, the preprocessor ignores such code snippet and it is not compiled.

Conditional compilation directives such as `#ifdef` and `#endif` encompass code associated with optional and alternative features, for example. Then, to remove a feature from the final product, we set to false the directive tag correspondent to the feature. Figure 2.5 illustrates a snippet of the *arena* feature implemented using conditional compilation.

Conditional compilation consists of a very simple programming model. There is no need to learn new languages. Besides, conditional compilation addresses not only fine-grained variabilities, but also coarsed-grained ones, like when we encompass an entire class with an `#ifdef` directive. However, despite their widespread

use, preprocessors suffer of several drawbacks, including no support for separation of concerns [9, 12, 25]. In what follows, we provide an overview of some arguments against conditional compilation:

- **Comprehensibility:** we can potentially use many `#ifdef` directives—that might be nested—within a class or even within a single method. Additionally, the base code (the one without `#ifdefs`) and feature code may have dependencies which are difficult to locate and understand. For example, the base code may declare a variable and the feature code might use it. Mixing `#ifdef` directives with the source code complicates the task of understanding and even reading the code. This way, maintaining it is difficult as well. Therefore, maintenance tasks in one particular feature are problematic. Developers cannot focus on such a feature, since the other ones might distract them [12]. Besides, the total source lines of code (SLoC) sometimes increases significantly due to the `#ifdef` directives;
- **Separation of concerns:** a preprocessor-based implementation scatters feature code across the entire base code, leading to tangling and traceability problems [12]. This way, instead of looking at one single module, we need to search in many code artifacts for snippets of the feature we are maintaining. In addition, feature tangling distracts the developer, as mentioned. Usually, it is difficult to focus on one particular feature;
- **Sensitivity to subtle errors:** when using conditional compilation, developers are prone to introduce subtle errors [12]. For instance, they can annotate an opening bracket but not the closing one. And this situation gets even worse: compilers are not able to detect this syntactic error unless the developer eventually tries to build the system with the problematic variant. One can build all variants to detect problems like these. However, depending on the number of features, this ranges from costly to prohibitive. For example, a small SPL with no feature constraints and 10 optional features that can be arbitrarily combined requires $2^{10} = 1024$ distinct products to be compiled.

Next, we present an approach that aims at reducing these drawbacks.

2.1.2 Virtual Separation of Concerns

Virtual Separation of Concerns (VSoC) [11] allows developers to hide feature code not relevant to the current task, being important to reduce some of the preprocessors drawbacks. Using this approach, developers can, to some extent, maintain a feature without being distracted by the others, aiding comprehensibility. Figure 2.6 shows the *arena* feature hidden.

In this context, `#ifdef` directives are no longer needed. For feature annotations, developers rely on background colors, so that code fragments belonging to a feature are shown with a background color. For our example, the background color associated with *arena* is gray. VSoC relies on tools for hiding and coloring the feature code. One such tool is the Colored IDE (CIDE) [11].

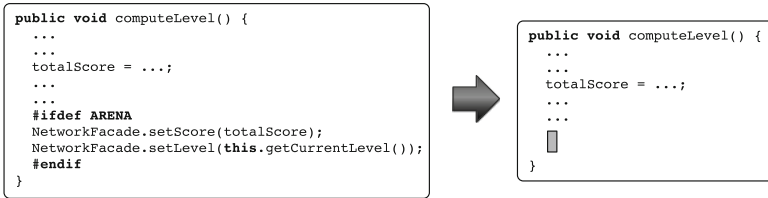


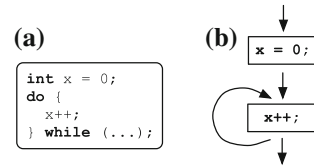
Fig. 2.6 From conditional compilation to VSoC: arena feature hidden

Now, we revisit each problem discussed in Sect. 2.1.1. The idea consists of discussing the conditional compilation drawbacks that VSoC minimizes.

- Comprehensibility and separation of concerns:** when using the VSoC approach, it is possible to separate concerns by using views. Technically, this is done by hiding the code not related to the feature we are maintaining. The approach is called virtual since the hidden code is still there. This way, VSoC is purely annotative: there is no physical code extraction to other modularization units such as aspects [13]. Besides, when using the VSoC approach, we abandon `#ifdef` directives, avoiding code pollution and decreasing the system size in terms of SLoC. We employ background colors, instead;
- Sensitivity to subtle errors:** to avoid problems of wrong annotations, VSoC allows only disciplined feature annotations. To define disciplined annotations, we use the following definition [16]: “annotations on one or a sequence of entire functions are disciplined. Furthermore, annotations on one or a sequence of entire statements are disciplined. All other annotations are undisciplined.” So, we can annotate program elements such as entire classes, methods, and statements. However, we cannot annotate, for instance, an `if` statement and its opening bracket without the closing one. In such a case, the annotation is undisciplined, and the tool does not allow it. Also, we cannot annotate not optional nodes—like the return type of a method—because it would invalidate the AST [11]. This approach limits the expressive power of annotations in exchange for avoiding syntax errors. In addition, CIDE provides a product-line-aware type system. The idea consists of checking if the variants are well-typed. For example, CIDE shows an error when we annotate a variable declaration but not the variable use; or if we annotate the only `return` statement of a method.

Despite reducing some drawbacks, developers are still not aware of feature dependencies. Next, we present dataflow analysis, the technique we use to capture these dependencies throughout this work.

Fig. 2.7 A simple example program and its corresponding control-flow graph. **a** An example program. . . , **b** . . . and its CFG



2.2 Dataflow Analysis

In this section, we review dataflow analysis. We present the three constituents of dataflow analysis (control-flow graph, lattice, and transfer functions) in Sect. 2.2.1. Then, in Sect. 2.2.2, we show one type of dataflow analysis named reaching definitions.

2.2.1 The Three Constituents

The three constituents of a dataflow analysis [14] are:

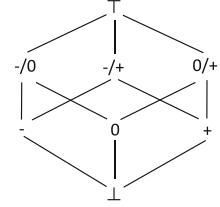
- a *control-flow graph* (on which we perform the analysis);
- a *lattice* (representing values of interest during the analysis); and
- *transfer functions* (that simulate the program execution at compile-time).

In what follows, we recall each of these three constituents (Sects. 2.2.1.1, 2.2.1.2, and 2.2.1.3) and describe how we may combine them to analyze an input program (Sect. 2.2.1.4) [5, 6].

2.2.1.1 Control-Flow Graph

Conceptually, a control-flow graph (CFG) is the abstraction of an input program on which a dataflow analysis runs. A CFG is a directed graph where the nodes are the statements of the input program and the edges represent flow of control according to the semantics of the programming language. Some analyses use *forward* flow and others use *backwards* flow. The difference is that we reverse the directions of the arrows. Figure 2.7a depicts a tiny program fragment and its corresponding control-flow graph (Fig. 2.7b). We build a control-flow graph inductively from the syntactic structure of the program, although exceptions and virtual dispatching may complicate this process. An analysis may be *intraprocedural* or *interprocedural*, depending on how functions are handled in the CFG.

Fig. 2.8 A Hasse diagram of a lattice for sign analysis



2.2.1.2 Lattice

In dataflow analysis, we carefully arrange the information calculated by the analysis in a lattice, $\mathcal{L} = (D, \sqsubseteq)$ where D is a set of elements and \sqsubseteq is a *partial-order* on the elements.

Lattices are usually and conveniently described diagrammatically using so-called *Hasse Diagrams* as in Fig. 2.8, which depicts a lattice for analysing the sign of an integer. Each element of the lattice captures information of interest by the analysis we are executing. For example, “+” represents the fact that the value is always positive, “0/-” that the value is always either zero-or-negative. Moreover, a lattice often has two special elements; \perp at the bottom of the lattice which usually means “not analyzed yet” whereas \top at the top of the lattice usually means “we don’t know” the value at compile-time.

The partial ordering of the elements is depicted using the convention that $x \sqsubseteq y$ if and only if x is depicted below y in the diagram (according to the lines of the diagram). For example, $\perp \sqsubseteq +$ (since \perp is directly below $+$) and $0 \sqsubseteq \top$ (since 0 is transitively below \top), whereas $- \not\sqsubseteq 0$ and $- \not\sqsubseteq 0/+$.

In this context, the order is important during analysis as it induces a *least upper bound* operator, \sqcup , on the lattice elements which we use to combine information in the analysis when control-flows meet. For instance, $\perp \sqcup 0 = 0$, $0 \sqcup + = 0/+$, and $- \sqcup 0/+ = \top$.

Notice that the lattice we present arranges signs of an integer. However, the information the lattice should deal with depends on the analysis we intend to perform. For instance, if we need to analyze the value of a variable x , the lattice would arrange assignments to that variable. To better explain, consider the left-hand side of Fig. 2.9, which illustrates another toy program containing assignments to the variable x . The right-hand side presents the lattice. Not surprisingly, $x = 0$ and $x = 9$ indicate that x is assigned to 0 and 9, respectively.

As explained, \top represents the fact that “we don’t know” the value of x at compile time. Therefore, x is assigned to either 0 or 9. This situation happens right before the call to method m , exactly where the control-flows meet. During the analysis, we combine information by using the least upper bound operator: $\{x = 0\} \sqcup \{x = 9\} = \top$.

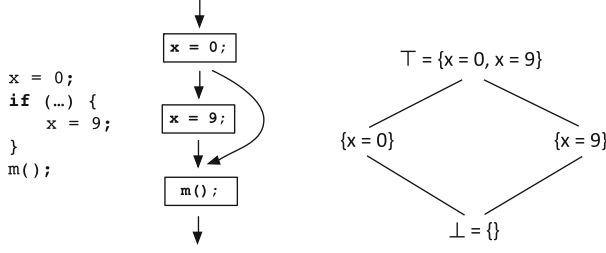


Fig. 2.9 Lattice arranging variable assignments

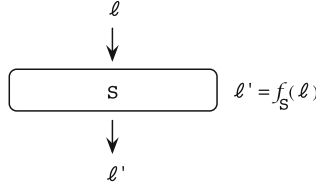


Fig. 2.10 Illustration of the effect of transfer function, f_S

$$f_{x=0}(\ell) = 0 \qquad f_{x++}(\ell) = \begin{cases} \top & \ell \in \{-/+, -/0, \top\} \\ + & \ell \in \{0, +, 0/+\} \\ -/0 & \ell = - \\ \perp & \ell = \perp \end{cases}$$

Fig. 2.11 Transfer functions for $x = 0$ and $x++$

2.2.1.3 Transfer Functions

In a dataflow analysis, each statement, S , of the program has an associated transfer function, $f_S : D \rightarrow D$, which simulates execution of S at compile-time (with respect to what is being analyzed). Figure 2.10 illustrates the effect of executing transfer function f_S ; lattice element, ℓ , flows into the statement node, the transfer function computes $\ell' = f_S(\ell)$, and the result, ℓ' , flows out of the node.

As an example, Fig. 2.11 illustrates transfer functions for two assignment statements. We use these functions for analyzing the sign of variable x using the sign lattice in Fig. 2.8

Now, we detail each function. The first one, $f_{x=0}$, is the constant zero function capturing the fact that x will always have the value zero after executing $x = 0$. The transfer function, f_{x++} , simulates the execution of $x++$; e.g., if x was negative ($\ell = -$) prior to execution, we know that its value after execution will always be negative-or-zero ($\ell' = -/0$).

Analogously, we have transfer functions when dealing with other type of information [26], like when the lattice arranges variable assignments. For example, the transfer function $f_{x=9}$ simulates the execution of the $x = 9$ statement.

Fig. 2.12 Applying the transfer function for the $x = 9$ statement

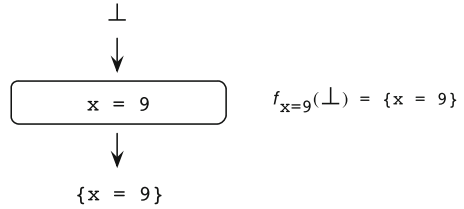


Figure 2.12 depicts that before executing the assignment, we do not know the value of x . Then, the \perp element flows into the statement and we execute function $f_{x=9}(\perp) = \{x = 9\}$. It is easy to notice that now we know the value of x , i.e., $x = 9$.

Depending on the analysis, the transfer function that deals with assignments is a bit more complicated, as we shall see in Sect. 2.2.2, where we illustrate the transfer function for the reaching definition analysis.

2.2.1.4 Analyzing a Program

Figure 2.13 illustrates how we combine the three constituents to perform dataflow analysis of the input program from Fig. 2.7a. In this context, we first build a control-flow graph (Fig. 2.7b) and annotate with program points which are the *entry* and *exit* points of the statement nodes of the CFG (depicted as gray circles in Fig. 2.13a). In our example, there are four points labelled with the letters a to d . Second, we turn the annotated CFG into a *whole-program transfer function*, $T : D^4 \rightarrow D^4$, which works on four copies of the lattice, \mathcal{L} , since we have four program points (a to d). The entry point, a , is assigned an initialization value which depends on the analysis (for the sign analysis it is bottom, $a = \perp$). For each of the program points, we capture the effect of the program using the statement transfer functions for simulating the effect of statements (e.g., $b = f_{x=0}(a)$). Also, we use the least-upper bound operator to combine flows (for instance, $c = b \sqcup d$). For our tiny program, the whole-program transfer function becomes:

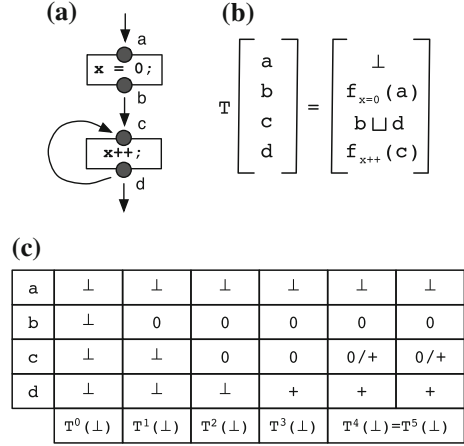
$$T((a, b, c, d)) = (\perp, f_{x=0}(a), b \sqcup d, f_{x++}(c))$$

Then, we use the Fixed-Point Theorem [20] to compute the fixed-point of the function, T , by computing $T^i(\perp)$ for increasing values of i (see the columns of Fig. 2.13c), until nothing changes.

Figure 2.13c shows that we reach the fixed-point in five iterations (see $T^4(\perp) = T^5(\perp)$). Therefore, the result of the analysis is:

$$T((\perp, 0, 0/+, +)) = (\perp, 0, 0/+, +)$$

Fig. 2.13 An input program is turned into a CFG annotated with program points (here, a , b , c , and d); which gives rise to a whole-program transfer function, $T : D^4 \rightarrow D^4$, for which we compute the least fixed point by computing $T^i(\perp)$ for increasing i , until nothing changes. **a** CFG. **b** Whole-program transfer function, T . **c** Fixed-point iteration



This means that “ $a = \perp, b = 0, c = 0 / +, d = +$ ” is the unique least fixed-point of T . So, we can deduce that the value of the variable x is always zero at program point b , it is zero-or-positive at point c , and positive at point d .

2.2.2 Reaching Definitions

In this section we present a common dataflow analysis named reaching definitions. The analysis idea consists of determining statically which variable definitions can reach a given program point p . In this context, every assignment to a variable x is a definition. This way, a definition d reaches a point p if there exists a path from the point immediately following d to p such that d is not killed (redefined) along that path [1]. We kill the definition of x if there exists any other assignment to x in another point along the path.

We summarize these concepts by using an example. Figure 2.14a shows a definition in the beginning of the program: $tS = p$. As Fig. 2.14b depicts, this definition does not reach point 6, since we kill $tS = p$ in all paths from the definition to this point. $tS = x$ kills $tS = p$ in path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$. Analogously, $tS = y$ kills $tS = p$ in path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$. On the other hand, the definition $tS = y$ reaches point 6 in path $4 \rightarrow 5 \rightarrow 6$. No definition in this path kills $tS = y$.

Reaching definitions are used to compute *use-def* and *def-use* graphs. The *def-use* graph is similar to a CFG, except that edges go from definitions to possible uses [23]. The *use-def* is similar but goes in the opposite way. Figure 2.14c depicts the *def-use* graph of our small program. These graphs are the basis for compiler optimizations, such as dead code elimination, code motion, and constant propagation [23].

When using reaching definitions, we compute analysis information with respect to assignments. This way, we store them in the lattice. Now, we discuss the transfer

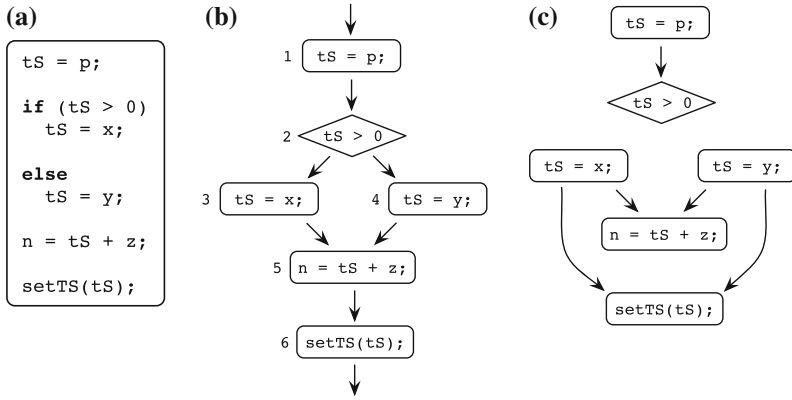


Fig. 2.14 According to the reaching definitions analysis, $tS = p$ does not reach point 6; reaching definitions are also useful to generate def-use graphs. **a** Program. **b** CFG. **c** Def-use graph

functions for reaching definitions. In this context, when executing the following statement, where d means definition,

$$d : tS = x;$$

we “generate” the definition of variable tS . At the same time, we “kill” the other definitions of this variable. This way, we might express our transfer function, f_d , as:

$$f_d(in) = gen_d \cup (in - kill_d)$$

where gen_d is the set of definitions generated by d ; $kill_d$ is the set of definitions that d kills; and in is the set of definitions that flows into the definition d we are analyzing. Taking the following program as an example

$$\begin{aligned}
d_1 : x &= 1; \\
d_2 : y &= 2; \\
d_3 : x &= 2; \\
d_4 : a &= x + y;
\end{aligned}$$

and applying the transfer function for definition d_3 , we have:

$$\begin{aligned}
f_{d_3}(in) &= gen_{d_3} \cup (in - kill_{d_3}) \\
f_{d_3}(in) &= \{x = 2\} \cup (\{x = 1, y = 2\} - \{x = 1\}) \\
f_{d_3}(in) &= \{x = 2\} \cup \{y = 2\} \\
f_{d_3}(in) &= \{x = 2, y = 2\}
\end{aligned}$$

Therefore, the assignment $x = 1$ does not reach the point after d_3 , since we killed it due to $x = 2$.

```

protected void flowThrough(FlowSet source, Unit unit, FlowSet dest) {
    if (unit instanceof AssignStmt) {
        AssignStmt assignment = (AssignStmt) unit;
        kill(source, assignment, dest);
        gen(dest, assignment);
    } else {
        source.copy(dest);
    }
}

```

Fig. 2.15 Transfer function for the reaching definitions analysis implemented in SOOT

To better illustrate this transfer function, we now present an implementation based on SOOT [24], an optimization framework for analyzing Java programs. To implement intraprocedural analyses in SOOT, we typically extend either the `ForwardFlowAnalysis` or the `BackwardFlowAnalysis` classes. Because the reaching definitions analysis uses the forward flow, we extend the former class. Then, we implement some template methods such as `merge` (the least upper bound operator), `copy` (responsible for copying lattice elements), and `flowThrough`, which represents the transfer function. Figure 2.15 illustrates the `flowThrough` method.

Notice that this method takes an `Unit` object as argument, representing the statement in which the transfer function will take place [26]. Also, it has two more arguments: `source` and `dest`. The first one is the lattice flowing into the transfer function. In case the statement we are applying the function is an assignment (in SOOT, an instance of the `AssignStmt` class), we update the lattice accordingly into the `dest` object by using the `gen` and `kill` functions. Otherwise, there is nothing to do. We simply copy the `source` lattice into the `dest` (see the `else` statement in Fig. 2.15).

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques, and Tools*, 2nd edn. Addison-Wesley, Boston (2006)
2. Alves, V., Matos P. Jr., Cole, L., Borba, P., Ramalho, G.: Extracting and Evolving Mobile Games Product Lines. In: *Proceedings of the 9th International Software Product Line Conference (SPLC)*, Lecture Notes in Computer Science, vol. 3714, pp. 70–81. Springer (2005)
3. Alves, V.: *Implementing Software Product Line Adoption Strategies*. Ph.D. thesis, Federal University of Pernambuco (2007)
4. Batory, D.: Feature models, grammars, and propositional formulas. In: *Proceedings of the 9th International Conference on Software Product Lines (SPLC)*, pp. 7–20. Springer, New York (2005)
5. Brabrand, C., Ribeiro, M., Tolêdo, T., Borba, P.: Intraprocedural dataflow analysis for software product lines. In: *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 13–24. ACM (2012)
6. Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., Borba, P.: Intraprocedural dataflow analysis for software product lines. *Lecture Notes in Computer Science: Transactions on Aspect-Oriented Software Development I*, pp. 73–108. Springer (2012)

7. Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston (2002)
8. Concurrent Versions System. <http://savannah.nongnu.org/projects/cvs/> (2011)
9. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.* **28**, 1146–1170 (2002)
10. Kang, K.-C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.: *Feature-Oriented Domain Analysis (FODA). Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute (1990)
11. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pp. 311–320. ACM (2008)
12. Kästner, C., Apel, S.: Virtual separation of concerns—a second chance for preprocessors. *J. Object Technol.* **8**(6), 59–78 (2009)
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science, pp. 220–242 (1997)
14. Kildall, G.A.: A unified approach to global program optimization. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pp. 194–206. ACM (1973)
15. Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: A case study in refactoring a legacy component for reuse in a product line. In: *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pp. 369–378. IEEE Computer Society (2005)
16. Liebig, J., Kästner, C., Apel, S.: Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In: *Proceeding of the 10th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 191–202. ACM (2011)
17. van der Linden, F., Schmid, K., Rommes, E.: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin (2007)
18. Matos P. Jr.: *Analyzing techniques for implementing product line variabilities*. Master’s thesis, Federal University of Pernambuco (2008)
19. Pilato, C.M., Collins-Sussman, B., Fitzpatrick, B.W.: *Version Control with Subversion*, 1st edn. O’Reilly, Sebastopol (2004)
20. Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*. Springer, New York (1999)
21. Parnas, D.L.: On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–9 (1976)
22. Pohl, K., Bockle, G., van der Linden, F.J.: *Software Product Line Engineering*. Springer, Berlin (2005)
23. Schwartzbach, M.I.: *Lecture Notes on Static Analysis* (2008)
24. Soot: A Java Optimization Framework. <http://www.sable.mcgill.ca/soot/> (2010)
25. Spencer, H., Collyer, G.: #ifdef considered harmful, or portability experience with C news. In: *Proceedings of the Usenix Summer Technical Conference*, pp. 185–198. Usenix Association (1992)
26. Társis Tolêdo.: *Dataflow analysis for software product lines*. Master’s thesis, Federal University of Pernambuco (2013)
27. Trujillo, S., Batory, D., Diaz, O.: Feature refactoring a multi-representation program into a product line. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pp. 191–200. ACM (2006)

Emergent Interfaces for Feature Modularization

Ribeiro, M.; Borba, P.; Brabrand, C.

2014, XI, 84 p. 49 illus., Softcover

ISBN: 978-3-319-11492-7