A Semi-Automatic Approach for Extracting Software Product Lines

Marco Tulio Valente, *Member*, *IEEE Computer Society*, Virgilio Borges, and Leonardo Passos

Abstract—The extraction of nontrivial software product lines (SPL) from a legacy application is a time-consuming task. First, developers must identify the components responsible for the implementation of each program feature. Next, they must locate the lines of code that reference the components discovered in the previous step. Finally, they must extract those lines to independent modules or annotate them in some way. To speed up product line extraction, this paper describes a semi-automatic approach to annotate the code of optional features in SPLs. The proposed approach is based on an existing tool for product line development, called CIDE, that enhances standard IDEs with the ability to associate background colors with the lines of code that implement a feature. We have evaluated and successfully applied our approach to the extraction of optional features from three nontrivial systems: Prevayler (an in-memory database system), JFreeChart (a chart library), and ArgoUML (a UML modeling tool).

Index Terms—Software product lines, virtual separation of concerns, refactoring tools, annotations.

1 Introduction

Software product lines (SPL) have emerged as an important development approach to avoid one-off software implementations. Essentially, SPL-based development advocates the generation of programs by combining particular features (or variabilities) with a common set of core components [9], [42], [36], [45]. The ultimate goal is to create program families that meet the requirements of a particular market segment. For example, in the case of database systems, developers might need to derive tailored systems with or without transactions, with or without replication, with or without logging, and so on [23].

Organizations interested in adopting SPLs usually rely on an extractive approach, where existing software systems are used as a baseline for bootstrapping an SPL [28], [15]. However, despite the benefits of starting from an existing codebase, the extraction of an SPL is usually a time-consuming task. First, developers must identify the classes, methods, and fields responsible for the implementation of each feature. Next, they must manually locate the lines of code that reference the elements discovered in the previous step. Finally, depending on the implementation technology, they must extract these lines to independent modules (e.g., aspects) or annotate them in some way (e.g., using `#ifdef` and `#endif` preprocessor directives).

To speed up product line extraction, this paper describes a semi-automatic approach to annotate the code of optional features in SPLs. The proposed approach is based on an existing tool for software product line development called Colored IDE (CIDE) [24]. CIDE enhances the standard IDE with the ability to associate background colors with the lines of code in charge of implementing features. Therefore, instead of textual annotations, such as `#ifdef` and `#endif`, visual colors are used as annotations. For example, in the above-mentioned database system example, developers can mark all lines of code belonging to the feature *Replication* with the color red. Furthermore, CIDE supports the generation of projections of a system where the code annotated with a given color is not shown. Using this form of projection, developers can generate a software product without a particular feature.

The approach described in this paper leverages the CIDE tool by supporting the semi-automatic annotation of code related to optional features. For this purpose, SPL developers must initially supply a set of syntactic elements, including, for example, packages, classes, methods, and fields associated with the features under extraction. These elements are called feature seeds, or simply seeds. The annotation algorithm is a fixed-point algorithm with two phases. In the first phase, called color propagation, the program elements that reference the seeds are visually annotated with a color reported by the developer. For example, in our refactoring of the ArgoUML modeling tool, the method `debug` has been classified as one of the seeds for the feature *Logging*. Therefore, during the color propagation phase, every call to `debug` will be visually annotated with the color selected by the developer to denote *Logging*. In the second phase, called color expansion, the algorithm checks whether a color can be expanded to its enclosing lexical context. For example, suppose we have the following method:

```
void mySingleClick(int tab) {
  debug(''single: '' +···); // annotated
}
```

- *M.T. Valente is with the Department of Computer Science, Federal University of Minas Gerais, Av. Antônio Carlos, Belo Horizonte, MG 31270-010, Brazil. E-mail: mtov@dcc.ufmg.br.*
- *V. Borges is with the Department of Informatics, COTEMIG, Brazil. E-mail: virgilioborges@cotemig.com.br.*
- *L. Passos is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada. E-mail: lpassos@gsd.uwaterloo.ca.*

Manuscript received 23 Dec. 2009; revised 14 Aug. 2010; accepted 7 May 2011; published online 6 June 2011.
Recommended for acceptance by M.-A. Storey.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-12-0412.
Digital Object Identifier no. 10.1109/TSE.2011.57.

0098-5589/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

Because this method only calls `debug` (annotated in the first phase), the method's body—and associated calls—can also be annotated with the color that denotes *Logging*. The semantics in this case is that `mySingleClick` is not needed in products where *Logging* has been disabled (including its calls).

We have successfully applied the proposed approach to the extraction of three nontrivial SPLs. First, we have extracted five features from an in-memory database system called Prevayler. This system has been previously used in other studies about SPL extraction, using aspect-oriented and feature-oriented programming [31], [16]. However, in such experiences, many enabling transformations have been applied to the original Prevayler code, mainly to associate feature-related code with the specific program locations that can be extended by languages such as AspectJ [27] and Jak [3]. On the other hand, our annotation algorithm does not impose any constraints on the object-oriented code.

In a second case study, we have extracted three features from the JFreeChart library. In this study, using regular expressions to describe the seeds of each feature, we have been able to annotate 99 percent of the feature's source code (when compared with a manual extraction performed independently by one of the paper's authors). In the last study, we extracted four features from the ArgoUML modeling tool. By using the entire packages responsible for the feature's implementation as seeds, we have achieved recall rates ranging from 88 to 97 percent (when compared with a manual refactoring conducted independently by a developer with no knowledge about our annotation approach).

The organization of the paper is as follows: We start with an overview of the CIDE tool (Section 2). Next, Section 3 describes the rationale behind the proposed approach, including a description of its central goals, underlying semantics, and limitations. Section 4 details the proposed annotation algorithm, including a formal presentation of the color propagation and color expansion phases. This section also presents the semi-automatic expansions suggested by the algorithm when it is not trivial to expand a color to its enclosing context. Section 5 introduces the extraction process that developers should follow to apply the proposed annotation algorithm. Section 6 presents the results of the application of the proposed process to extract optional features from three nontrivial systems (Prevayler, JFreeChart, and ArgoUML). Section 7 discusses related work, and Section 8 concludes the paper with a summary of the its main contributions and a discussion of future work.

2 CIDE: Colored IDE

Proposed by Kästner and Apel, CIDE is a tool that leverages conventional preprocessors by replacing textual directives by visual annotations in the form of background colors [24], [26]. As presented in Fig. 1, visual annotations are exhibited directly in the editor provided by the Eclipse IDE. For example, in this figure, the code fragments related to synchronization concerns have been annotated using the color blue and the fragments related to logging have been marked in yellow. As can be observed in the figure, visual annotations reduce the code obfuscation problem usually

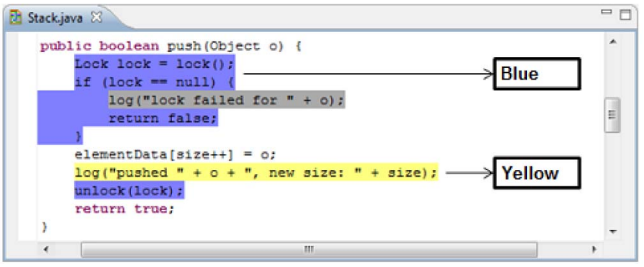

Fig. 1. CIDE screenshot.

observed when using textual directives, such as `#ifdef` and `#endif` [41], [14].

To facilitate navigation over the annotated code, the tool allows developers to jump between the code fragments associated with a certain feature. Moreover, as in the folding and unfolding of program elements provided by conventional IDE, developers can hide the code associated with a given feature. In this way, they can synthesize a product without this feature. To avoid the syntax errors that can occur at derivation time when using `#ifdef` and `#endif`, CIDE supports a disciplined form of annotation. Essentially, it is not possible to mark every token of the target program (including tokens that do not carry semantic information, such as commas or brackets). Instead, only coarse-grained structures, such as complete declarations, definitions, and statements, can be annotated.

To handle overlapping features, CIDE mixes the respective background colors (e.g., logging tangled with replication will receive both the color assigned to replication and the color assigned to the logging feature). Background colors do not scale to SPLs with hundreds of features (because the human eye can only discern a limited number of colors). However, the background colors are sufficient to distinguish nonmarked code from the code associated with features. Furthermore, tooltips show the names of the features associated with any marked code. Finally, CIDE provides a product-line-aware type system that guarantees that all variants of an SPL are well typed [22]. For example, CIDE's type checker can detect situations like dangling method calls (i.e., methods that are removed in a variant, but that are still called). When a type error is detected, the tool suggests a list of so-called "quick fixes," which are usually restricted to the annotation of the ill-typed statement. Therefore, the suggested repair actions do not scale to the whole program (as in the propagation phase of the algorithm proposed in this paper).

2.1 CIDE versus Other SPL Extraction Approaches

Table 1 summarizes the differences between CIDE and three well-known approaches for extracting variabilities: conditional compilation (CC), aspect-oriented programming (AOP), and feature-oriented programming (FOP). In this comparison, we have considered AspectJ [27] as being the representative of the various AOP techniques and Jak [3] as being the representative of FOP. With respect to the properties considered in Table 1, feature cohesion denotes the capability to manipulate features under a cohesive code infrastructure [32]. For example, AspectJ does not allow developers to assign a name to all program extensions and to manipulate them as a single module.

TABLE 1
Comparing CIDE with CC, AOP, and FOP

Properties	CC	AOP	FOP	CIDE
Extensions	Fine-grained	Coarse-grained	Coarse-grained	Fine-grained
Separation Concerns	No	Yes (physical)	Yes (physical)	Yes (virtual)
Feature cohesion	No	No	Yes (physical)	Yes (virtual)

In contrast, Jak relies on mixin-layers and collaborations to provide feature cohesion. Finally, to emulate feature cohesion, CIDE provides support for editable views, i.e., it is possible to configure the editor to show only the code of one or more features (and some necessary context), while hiding everything else. However, when updating a view, developers should be aware that the view's code may interfere with and suffer interference from the code context hidden by the view.[1]

3 Proposed Approach

The central goal of our approach is to automate the extraction of optional features from an existing system. We assume that a system can be decomposed into mandatory features, which constitute the system's core, and optional features, whose implementation crosscuts the core. For optional features, our approach relies on CIDE to annotate the blocks of code that contribute to the feature's implementation. For example, suppose that an optional feature F is associated with a color c. In this case, the semantics behind the annotation of a block of code S with color c is as follows: To build a product without F, we must remove S. In summary, because the goal is to disable features from an existing codebase, our approach is based on removing code (or annotating code, according to CIDE's semantics).

The inputs of the annotation algorithm are a set of program elements S associated with a given feature F and a color c. Elements of S are called feature seeds, or simply seeds. The following program elements can be used as seeds: packages, classes or interfaces, methods, and fields. It is also possible to use regular expressions to define seeds (e.g., `Logger.*` will match any method or field from the class `Logger`). As an output, the algorithm applies the background color c to the code fragments associated with the optional feature F.

3.1 Optional Features

Our approach targets the extraction of optional features, i.e., features that can be safely removed without disrupting the behavior of the core. In other words, an optional feature can be plugged-in or plugged-out from the system. When it is plugged-in, it complements the system with its functionality (e.g., logging). When it is plugged-out, the system is still operational (e.g., a system without logging is useful in many scenarios). More formally, suppose the following flow of execution of an existing program:

$$\text{main} \longrightarrow I_1 \longrightarrow F \longrightarrow I_2 \longrightarrow B,$$

where B denotes a block of code from the base program, F denotes a block of code marked as related to the optional feature F, I_1 denotes the intermediary code executed between the start of the program and F, and I_2 denotes the intermediary code executed between F and B. Moreover, suppose that $pre(B)$ and $pos(B)$ are Boolean expressions denoting the pre and postconditions expected by B, respectively. A feature F is optional if for any execution flow like the previous one, the following execution is also possible:

$$\text{main} \longrightarrow I_1 \longrightarrow I_2 \longrightarrow \mathbf{assert}(pre(B)) \longrightarrow B \longrightarrow \mathbf{assert}(pos(B)).$$

In other words, the removal of F must not interfere with the correctness of B, including its pre and postconditions. Furthermore, the removal of F must not syntactically affect B.

3.2 Feature Dependencies

A feature F_2 depends on a feature F_1 if it is impossible to have a product with F_2 selected and F_1 unselected. For example, in the product line we extracted for ArgoUML, the feature *ActivityDiagram* depends on the feature *StateDiagram*. In fact, the code that implements *ActivityDiagram* relies on code originally designed to implement *StateDiagram* (e.g., there are classes associated with *ActivityDiagram* that extend classes associated with *StateDiagram*).

More formally, a feature F_2 depends on a feature F_1 if there is at least an execution flow in the following form:

$$\text{main} \longrightarrow I_1 \longrightarrow F_1 \longrightarrow I_2 \longrightarrow F_2,$$

that fails after removing F_1, i.e.,

$$\text{main} \longrightarrow I_1 \longrightarrow I_2 \longrightarrow \mathbf{assert}(pre(F_2)) \longrightarrow \text{fail}.$$

Moreover, in the cases when the program does not compile after the removal of F_1, it is also assumed that F_2 depends on F_1. The dependency means that it is impossible to have a product with F_2 selected and F_1 unselected. However, our algorithm cannot detect and avoid the generation of products configured in this way. In other words, our assumption is that feature dependencies are discovered, mapped, and handled externally by developers, possibly using a feature management tool (such as pure::variants [4] or fmp [2]). Moreover, our approach does not tackle the possible impact of optional features in a user interface's layout or in external resources manipulated by the system (including databases, configuration files, build files, and so on).

3.3 Optional Feature Problem

In the extraction of SPLs, the optional feature problem designates the impossibility of implementing certain optional features in single modules [31], [25]. For example, the

1. For example, the view may change the value or even remove the declaration of a local variable that is used outside the view. As a second example, the view can rely on a value that is defined in an outside context.

TABLE 2
Stack Features and Seeds

Feature	Description	Seeds
Logging	Logs internal stack events	Class `Log`
Snapshot	Saves a snapshot in disk of the stack elements	Method `snapshot`
Replication	Replicates the stack in another server	Method `replicate`
Multithreading	Thread-safe stack	Class `Lock`

implementation of logging may be nested inside the implementation of another optional feature, such as replication. Therefore, code related to logging must be decomposed into modules such as `Core-Logging` (which provides logging to the system core) and `Replication-Logging` (which provides logging to the replication modules). In composition-based approaches, such smaller modules are called derivatives [31]. On the other hand, in the annotation-based algorithm proposed in this paper, the optional feature problem is tackled by assigning two colors to the code fragments associated with both features. Nevertheless, the semantics behind annotations does not change in such cases, i.e., to build a product without F, any code S marked with F's color must be removed (despite the existence of other colors assigned to S).

Example. Suppose a class `Stack`—adapted from [24]—with the optional features and respective seeds as described in Table 2. Fig. 2 shows the method `push` from this class after the annotations introduced by our approach (marked lines are indicated by comments). Initially, for *Logging*, our algorithm will annotate the method call in line 4. For the features *Snapshot* and *Replication*, the algorithm will propagate the respective colors to the calls in lines 11 and 13. After this propagation, the body of an `if` statement will be marked in both cases, but not the corresponding expressions. Therefore, a color expansion will be automatically applied to mark the complete `if` statements (lines 10-11 and 12-13).

For *Multithreading*, our algorithm will first annotate the declaration of the local variable `lk` (line 2) and its use in the `if` expression (line 3). In this case, however, it is not safe to infer that the annotation should be expanded to the whole `if` expression (as it depends on the return of the method `lock()`). Therefore, a semi-automatic expansion will be triggered in this case, i.e., the algorithm will ask the users whether they authorize the color expansion throughout the whole `if` expression. In this particular case, the suggested expansion will be accepted and the `if` expression will be annotated. Finally, this semi-automatic annotation will trigger an automatic expansion to mark the whole `if` statement (lines 3-6) as related to *Multithreading*. After such expansions, the logging call in line 4 will be marked with two background colors (*Multithreading* and *Logging*). Therefore, this call will be removed in any products with *Multithreading* disabled. It will also be removed in products with *Logging* disabled (regardless of whether *Multithreading* is enabled).

Finally, the feature *Replication* depends on the feature *Snapshot* (because replication relies on the data persisted by snapshot to send a copy to another host). However, our approach does not detect or avoid the generation of products with *Replication* enabled but with *Snapshot* disabled. As mentioned, the assumption is that the feature dependencies are managed externally by SPL engineers.

```
 1: boolean push(Object o) {
 2:   Lock lk = new Lock();              // M
 3:   if (lk.lock() == null) {          // M
 4:     Log.log("lock failed for " + o); // M,L
 5:     return false;                   // M
 6:   }                                 // M
 7:   elements[top++]= o;
 8:   size++;
 9:   lk.unlock();                      // M
10:   if ((size % 10) == 0)             // S
11:      snapshot("stack.db");          // S
12:   if ((size % 100) == 0)            // R
13:      replicate("stack.db", "srv2"); // R
14:   return true;
15: }
```

Fig. 2. Method `push` with features annotated by our approach (M = Multithreading; L = Logging; S = Snapshot; R = Replication).

4 Annotation Algorithm

Fig. 3 presents the algorithm's main routine. Essentially, it is a fixed-point algorithm that propagates the background color c throughout the program until all program elements that depend on the seeds S are marked. The algorithm has two phases: color propagation and color expansion. During the color propagation phase, the AST (Abstract Syntax Tree) nodes C_1 that correspond to the seeds S are annotated; next, the AST nodes C_2 that correspond to the elements defined in C_1 are also annotated and successively, until no new nodes can be annotated. The goal of the second phase, color expansion, is to check whether the enclosing context of the AST nodes annotated in the previous phase can also be marked.

Tables 3 and 4 present the auxiliary functions used in the color propagation and expansion phases. The functions from Table 3 return structural information about the target system, by querying its AST, and the functions from Table 4 return a set of AST nodes. For example, `call(m)` returns the AST nodes containing calls to the method `m`. Finally, the function `cide(t,c)` is used to annotate the AST nodes t with color c.

$$Main(Seed\ S, Color\ c) = \\ \forall s \in S \rightarrow ColorPropagation(s, c); \\ ColorExpansion();$$

Fig. 3. Main routine.

TABLE 3
AST Query Functions

Function	Returns
$classes(p)$	classes defined in package a p
$interfaces(p)$	interfaces defined in a package p
$meths(t)$	methods defined in a class or interface t
$fields(t)$	fields defined in a class t
$hasType(t)$	local variables and formal parameters of type t
$hasReturnType(t)$	methods with return type t
$impl(i)$	classes that implement the interface i (or an interface that extends i)
$extends(t)$	classes that extend the class t
$formal(p)$	formal parameters associated to the actual parameter p

4.1 Color Propagation

Fig. 4 presents the color propagation algorithm. As mentioned above, this algorithm marks all program elements that depend directly or indirectly on the seed S. More specifically, the program elements that can be affected by color propagation include packages (rule P1), classes (rule P2), interfaces (rule P3), methods (rule P4), fields (rule P5), local variables (rule P6), and formal parameters (rule P7). Next, we describe each of these rules.

- Rule P1: To annotate package p, we must propagate the color to the classes and interfaces declared in this package (lines 2-3). We must also annotate the `import` statements that include classes from p (line 4).
- Rule P2: To annotate class t, we must propagate the color to its methods and fields (lines 6-7), as well as to the following program elements that rely on t: subclasses (line 8), local variable declarations (line 9), methods returning this type (line 10), object instantiations (line 11), and `import` statements (line 12).
- Rule P3: To annotate interface i, we must propagate the color to its declaration (line 14), as well as to the following program elements that rely on i: implementation of the methods from this interface (line 15), local variable declarations (line 16), methods returning this type (line 17), and `import` statements (line 18).
- Rule P4: To annotate method m, we must propagate the color to its implementation (line 20), as well as to the calls to m (line 21) and to the methods that override m (line 22). It is important to mention that the algorithm only annotates calls that can be inferred statically. For example, consider a call `t.m()`, where the (static) type of the target object `t` is `T`. This call will get annotated in two situations:
  - when the implementation of `m` in `T` has been annotated;
  - when `m` is a method inherited from a superclass `T'` and the implementation of `m` in `T'` has been annotated.
- Rule P5: To annotate a field, we must propagate the color to its declaration (line 24), as well as to the program locations where the field is accessed (line 25).
- Rule P6: To annotate a local variable, we must propagate the color to its declaration (line 27), as well as to the program locations where the variable is read or updated (line 28).
- Rule P7: To annotate a formal parameter, we must propagate the color to its declaration (line 30), to the program locations where the parameter is accessed (line 31), and to the associated actual parameters (line 32).

The mechanics behind the proposed propagation rules resemble the transformations that must be applied in a program after removing a package (rule P1), a class (rule P2), an interface (rule P3), and so on. In fact, the main difference is that instead of removing code, the proposed rules call the CIDE tool to mark the code with the color associated with the feature under extraction.

TABLE 4
Functions that Return AST Nodes

Function	Returns
`call(m)`	calls to method m
`override(m)`	methods that override m in immediate subclasses
`access(x)`	read/write to field, local variable or formal parameter x
`actual(p)`	actual parameters associated to the formal parameter p
`declaration(x)`	declaration of a field, local variable, parameter, or interface x
`import(t)`	import statements for the type t
`import(p.*)`	import statements for types from package p
`new(t)`	instantiation of objects of type t
`impl(m)`	implementation of the method m
`package(p)`	implementation of the package p

$$
\begin{array}{lll}
1: & ColorPropagation(Package\ p, Color\ c) = & (P1)\\
2: & \quad \forall t \in classes(p) \rightarrow ColorPropagation(t, c);\\
3: & \quad \forall i \in interfaces(p) \rightarrow ColorPropagation(i, c);\\
4: & \quad \forall p = \texttt{import(p.*)} \rightarrow cide(p, c);\\
 & & \\
5: & ColorPropagation(Class\ t, Color\ c) = & (P2)\\
6: & \quad \forall m \in meths(t) \rightarrow ColorPropagation(m, c);\\
7: & \quad \forall f \in fields(t) \rightarrow ColorPropagation(f, c);\\
8: & \quad \forall s \in extends(t) \rightarrow ColorPropagation(s, c);\\
9: & \quad \forall v \in hasType(t) \rightarrow ColorPropagation(v, c);\\
10: & \quad \forall m \in hasReturnType(t) \rightarrow ColorPropagation(m, c);\\
11: & \quad \forall n = \texttt{new(t)} \rightarrow cide(n, c);\\
12: & \quad \forall p = \texttt{import(t)} \rightarrow cide(p, c);\\
 & & \\
13: & ColorPropagation(Interface\ i, Color\ c) = & (P3)\\
14: & \quad p = \texttt{declaration(i)} \rightarrow cide(p, c);\\
15: & \quad \forall t \in impl(i) \wedge \forall m \in meths(t) \wedge m \in i \rightarrow ColorPropagation(m, c);\\
16: & \quad \forall t \in hasType(i) \rightarrow ColorPropagation(t, c);\\
17: & \quad \forall m \in hasReturnType(t) \rightarrow ColorPropagation(m, c);\\
18: & \quad \forall p = \texttt{import(i)} \rightarrow cide(p, c);\\
 & & \\
19: & ColorPropagation(Method\ m, Color\ c) = & (P4)\\
20: & \quad p = \texttt{impl(m)} \rightarrow cide(p, c);\\
21: & \quad \forall s = \texttt{call(m)} \rightarrow cide(s, c);\\
22: & \quad \forall m' \in \texttt{overrides(m)} \rightarrow ColorPropagation(m', c).\\
 & & \\
23: & ColorPropagation(Field\ f, Color\ c) = & (P5)\\
24: & \quad d = \texttt{declaration(f)} \rightarrow cide(d, c);\\
25: & \quad \forall s = \texttt{access(f)} \rightarrow cide(s, c).\\
 & & \\
26: & ColorPropagation(LocalVariable\ i, Color\ c) = & (P6)\\
27: & \quad d = \texttt{declaration(i)} \rightarrow cide(d, c);\\
28: & \quad \forall s = \texttt{access(i)} \rightarrow cide(s, c).\\
 & & \\
29: & ColorPropagation(FormalParam\ p, Color\ c) = & (P7)\\
30: & \quad d = \texttt{declaration(p)} \rightarrow cide(d, c);\\
31: & \quad \forall s = \texttt{access(p)} \rightarrow cide(s, c);\\
32: & \quad \forall s = \texttt{actual(p)} \rightarrow cide(s, c).
\end{array}
$$

Fig. 4. Color propagation rules.

4.2 Color Expansion

Fig. 5 describes the color expansion algorithm. Essentially, the algorithm consists of a loop where automatic and semi-automatic expansions—detailed in Fig. 6 and Table 5—are sequentially called. The loop finishes when a fixed-point is reached, i.e., when the invoked rules do not insert any new annotation (regarding the previous iteration).

To propose the color expansion rules presented in Fig. 6, we have examined the program structures of Java composed of a block of code `C` that controls or creates a context for the execution of another block of code `S`. For example, `S` can denote the body of a loop statement and `C` can be the loop's expression. Once such program structures have been identified, we investigated whether color expansions are possible when only one of their parts has been annotated, i.e., an expansion from `S` (that has been annotated) to `C` (that has not been annotated) and vice versa. By following this process, the following color expansions have been proposed:

- Rule E1: The color used in the expression of a loop or conditional statement can be expanded to the statement's body. The rationale is that if the expression of an `if`, `while`, `switch`, etc., is not needed on a particular product, the whole statement can be removed in this product.
- Rule E2: The color used in the body of a loop or conditional statement can be expanded to the statement's expression if this expression does not produce side effects. The rationale is that if the body of an `if`, `while`, `switch`, etc., is not needed on a particular product and the expressions associated

$$
\begin{array}{l}
ColorExpansion() =\\
\quad \textbf{do}\\
\quad\quad oldcode = code;\\
\quad\quad BodyExpansion();\\
\quad\quad ExpExpansion();\\
\quad\quad StmExpansion();\\
\quad\quad MethExpansion();\\
\quad\quad ClassExpansion();\\
\quad\quad AssignExpansion();\\
\quad\quad SemiAutomaticExpansions();\\
\quad \textbf{while}\ code \neq oldcode
\end{array}
$$

Fig. 5. Color expansion.

$$
\begin{array}{ll}
BodyExpansion() = & (E1)\\
\quad s = [\texttt{if(exp) stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{while exp stm}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{do stm while exp}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) stm}] \wedge color(e_1, c) \wedge color(e_2, c) \wedge color(e_3, c) \rightarrow cide(s, c);\\
\\
ExpExpansion() = & (E2)\\
\quad s = [\texttt{if(exp) stm}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{while exp stm}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{do stm while exp}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{if(exp) }s_1\texttt{ else }s_2] \wedge color(s_1, c) \wedge color(s_2, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{case(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{switch(exp) \{stm\}}] \wedge color(\texttt{stm}, c) \wedge free(\texttt{exp}) \rightarrow cide(s, c);\\
\quad s = [\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{) stm}] \wedge color(\texttt{stm}, c) \wedge free(e_1, e_2, e_3) \rightarrow cide(s, c);\\
\\
StmExpansion() = & (E3)\\
\quad s = [\texttt{else stm}] \wedge color(\texttt{stm}, c) \rightarrow cide(s, c);\\
\quad s = [\texttt{return exp}] \wedge color(\texttt{exp}, c) \rightarrow cide(s, c);\\
\\
MethExpansion() = & (E4)\\
\quad s = [\texttt{t m(...) \{stm\}}] \wedge color(\texttt{stm}, c) \rightarrow ColorPropagation(m, c);\\
\\
ClassExpansion() = & (E5)\\
\quad s = [\texttt{class t \{ members \}}] \wedge color(\texttt{members}, c) \rightarrow ColorPropagation(t, c);\\
\\
AssignExpansion() = & (E6)\\
\quad s = [\texttt{i = exp;}] \wedge color(\texttt{i}, c) \rightarrow cide(s, c);
\end{array}
$$

Fig. 6. Color expansion rules.

with such statements are side effect free, then the whole statement can be removed in this product. In the specification of such rules, the function $free(\texttt{exp})$ tests whether an expression `exp` is side effect free. An expression has a side effect when it not only yields a value but also updates the program state. For example, an expression might modify a variable, read data from a file, or invoke a side-effecting function.

- Rule E3: The color used in the statements of an `else` must be expanded to include the clause. Similarly, the color used in the expression of a `return` must be expanded to include the statement.
- Rule E4: The color used in the body of a method must be expanded to its signature and therefore to the whole method body and its calls, which is implemented by calling a color propagation rule. The rationale is that if a method body is not needed on a particular product, we can completely remove this method.
- Rule E5: The color used by all members of a class must be expanded to its name and therefore to any

TABLE 5
Semi-Automatic Expansions

	Definition	Default Expansion
SE1	Only parts of an expression have been annotated with a color c	Annotate the whole expression with c
SE2	The `return` statements of a method have been annotated with a color c; but the method has other statements that have not been annotated with c	Annotate the whole method body with c
SE3	In a call to a method m, an actual parameter has been annotated with a color c; but the associated formal parameter has not	Annotate the whole method call with c
SE4	The right-hand side of an expression has been annotated with a color c; but the left-hand side has not	Annotate the left-hand side and its references with c
SE5	Color expansion E2 (Fig. 6) has not been applied (using color c) because it was not possible to infer whether `exp` is side effect free	Annotate `exp` with c

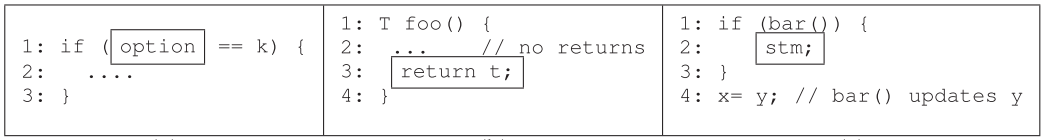

Fig. 7. Examples of program fragments where annotations can generate (a) syntax errors, (b) type errors, or (c) when it is not safe to apply a color expansion. A frame is used around the code annotated by the propagation and expansion phases.

use of the class, which is implemented by calling a color propagation rule.[2]

- Rule E6: The color used in the left-hand side of an assignment must be expanded to include its right-hand side. The rationale is that if it has already been established (or inferred) that variable `i` is not needed on a particular product, then assignments to `i` do not have any consequence, and they can be removed in the context of this particular product.

4.3 Semiautomatic Expansions

As illustrated in Fig. 7, the proposed algorithm can lead to code with syntactic or type errors. For example, in the program fragment presented in Fig. 7a, the proposed color propagation and expansion rules have marked only parts of a Boolean expression (line 1). Therefore, a projection of the system without the annotated code will present syntax errors. In the second example, the `return` statement of the method `foo` has been annotated (line 3). Consequently, after removing the annotated code, a type error will be reported (no `return` statement). Finally, Fig. 7c shows an example where it is not safe to apply an expansion from the body of an `if` statement (line 2) to the `if` expression (line 1). The reason is that this expression may generate side effects. In fact, the function `bar()` in the example updates the value of the variable `y`, which is used by the nonannotated code that succeeds the `if` statement (line 4).

In the above-mentioned situations, the application of any kind of automatic expansion is a challenging task. For example, it will depend on the value of `k` (Fig. 7a), on the computation performed by the `nonreturn` statements (Fig. 7b), or on a nontrivial analysis to detect whether `bar()` is a side effect free function (Fig. 7c).

When the inserted annotations lead to syntax/type errors (or when it is not trivial to expand an annotation), our CIDE extension generates a message explaining the problem. Moreover, we suggest a default expansion to handle the detected situation, as described in Table 5. For example, the tool can suggest annotating the whole expression—to avoid a syntax error like the one presented in Fig. 7a—or the complete method—to avoid a type error like the one presented in Fig. 7b. If the developer accepts the suggestion, the tool automatically applies it to the code. Finally, it is important to mention that semi-automatic expansions are only issued when there are no automatic expansions that can be applied.

As prescribed by expansion SE5, we deliberately left the decision on whether color expansions should capture expressions with side effects to developers. The reasons are twofold. First, detecting side-effects statically is by nature conservative (mainly in the presence of calls to methods provided by external libraries, dynamic method calls, native method calls, and so on [30]).[3] Second, some side effects can be benevolent or harmless. For example, they may only affect the feature's code without interfering with other features or with the core of the system.

5 Extraction Process

We implemented the proposed annotation algorithm on top of the CIDE tool. Our implementation assumes that developers follow the iterative process described in Fig. 8 to extract an SPL.

First, the developer must define the feature seeds. For this purpose, the developer should, for example, read the program's documentation, interview the developers who implemented the feature's code, browse the source code, study the program functionalities provided by the feature, etc. After choosing the seeds or at least the first candidates for seeds, the developer must execute our CIDE extension. During the execution, the tool may require feedback about semi-automatic expansions detected. In such cases, developers should decide whether the default expansions suggested by the tool should be accepted.

After the execution, the tool presents a report about the default expansions that have not been accepted. In this case, the developer should manually analyze the source code to fix the syntactic or type errors inserted when he has not accepted a default expansion.

Finally, the developer should evaluate whether the feature code has been completely marked. The degree of coverage achieved by the proposed algorithm depends on the defined seeds and semi-automatic actions. Essentially, the algorithm only guarantees that any code related to the feature that is statically reachable from the defined seeds will be marked. However, there may exist code fragments that are not reachable from the initially proposed seeds. To determine whether this situation has occurred, the developer can, for example, execute the system to check that any functionality associated with the feature has really been disabled. If there are relevant code fragments that have not been annotated, he must refine the seeds and repeat the process.

6 Evaluation

We investigated the feasibility of using our approach to annotate optional features in three medium-to-large systems. More specifically, our investigation aimed to answer the following research questions:

2. Therefore, color propagation and expansion are not sequential steps. By rules E4 and E5, color expansions can trigger color propagations, which in turn can enable new expansions and so on.

3. Our current implementation relies on a very conservative approach to detect side effects. The function `free(exp)`—called by rule E2 from Fig. 6—returns `true` only for expressions that do not have any method calls.

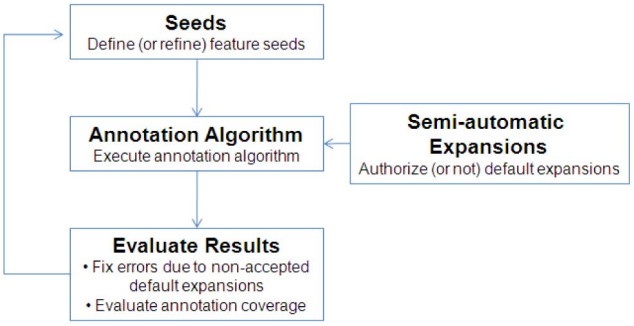

Fig. 8. Iterative extraction process.

1. How many iterations are needed in the extraction process to define seeds that lead to the annotation of features that are optional in the evaluated systems?
2. How many semi-automatic expansions are requested by the algorithm to annotate optional features in the evaluated systems?
3. When semi-automatic expansions are triggered, do developers generally accept the default actions suggested by the algorithm?
4. Can the proposed algorithm annotate precisely the parts of the code related to features that are optional in the evaluated systems? Specifically, we would like to answer whether the annotations generated by our approach correspond to code associated with a given feature (precision) and whether they cover all the code of this feature (recall).

In summary, we aim to answer whether our algorithm provides reasonable levels of precision and recall, while relying on simple feature seeds and requesting a limited number of semi-automatic expansions (preferably accompanied by meaningful default actions).

6.1 Target Systems

We have applied our approach to extract optional features in the following systems:

- Prevayler, a persistent framework for Java objects.[4] The monolithic, non-SPL-based version of the system has 2,974 (nonblanked, noncommented) LOC, 61 classes, 23 interfaces, 13 packages, and almost 162 KB of source code. The system has been used in other studies on the extraction of SPLs using AspectJ [16], [23] and AHEAD/Jak [31]. In this study, we decided to extract the following optional features: *Monitor* (supports logging of internal events), *Replication* (supports object replication between a server and multiple clients), *Censorship* (supports rollbacking in case of failures in transactions), *Snapshot* (supports database snapshots), and *GZip* (provides file compression support to snapshots).
- JFreeChart, a library that supports the display of a rich set of charts.[5] The library has 91,174 LOC, distributed over 504 classes, 103 interfaces, and 37 packages, corresponding to 7.15 MB of source code. In this study, we have extracted three optional features: *Pie Charts* (a particular kind of chart), *3D Charts* (an effect that crosscuts different charts), and *Statistical Charts* (such as histograms and box plots).
- ArgoUML, an open-source UML modeling tool.[6] The tool has 117,983 LOC, distributed over 1,637 classes, 93 interfaces, and 79 packages, corresponding to 8.95 MB of source code. In this study, we have extracted four features from the monolithic version of the system: *State Diagrams*, *Activity Diagrams*, *Logging*, and *Design Critics* (simple agents that continuously execute in a background thread of control, suggesting improvements in the UML design).

To select the features to be extracted in each system, we considered two types of features: relevant functional requirements (e.g., UML models in ArgoUML, charts in JFreeChart, and database operations in Prevayler) or typical nonfunctional requirements (such as logging in the Prevayler and ArgoUML experiments). Furthermore, we selected the *Design Critics* feature for ArgoUML due to its size and crosscutting behavior, which we noted by browsing the system's code before starting the experiment.

6.2 Study Design

We performed the following steps to answer the proposed research questions:

1. To provide a baseline for comparison, we decided to rely on a manual annotation of the code related to the features considered in the study. For Prevayler and JFreeChart, the manual extraction was conducted by one of the paper's authors. For ArgoUML—the most complex system considered in the study—we derived the baseline from an SPL-based implementation of the system using conditional compilation [10]. This implementation was independently constructed by a software developer with no knowledge of our algorithm.[7] To enable comparisons, we implemented a tool to automatically import preprocessor-based SPLs to CIDE. Essentially, this tool annotates the blocks of code delimited by `#ifdefs` with colors informed by the developer.[8]
2. In all cases, before starting the extraction, the developers studied the target system's architecture and documentation. Furthermore, after they considered the extraction finished, they also executed the system several times, to confirm that the features have been extracted (i.e., that their functionality has really been disabled, without any impact on the core's semantics). Table 6 provides information about the amount of source code annotated for each system and for the evaluated features. Observe that the features

4. http://www.prevayler.org, version 2.3.
5. http://www.jfree.org/jfreechart, version 1.0.13.
6. http://argouml.tigris.org, version 0.28.
7. The preprocessor-based implementation of a software product line for ArgoUML—called ArgoUML-SPL [10]—is available at http://argouml-spl.tigris.org.
8. CIDE supports the importing of SPLs implemented using the Antenna preprocessor (http://antenna.sourceforge.net). However, ArgoUML-SPL uses a preprocessor called javapp (http://www.slashdev.ca/javapp). Because the syntax of the directives defined by javapp and Antenna has minor differences, we have implemented our own importing tool.

TABLE 6
Bytes Annotated in the Manual Extraction

System	Size (MB)	Features	%Size
Prevayler	0.16	Monitor	4.0
		Replication	6.7
		Censoring	2.7
		Snapshot	6.9
		GZip	2.3
		Total	22.6
JFreeChart	7.15	Pie Charts	5.1
		3D Charts	2.3
		Statistical Charts	4.7
		Total	12.1
ArgoUML	8.95	State Diagrams	3.7
		Activity Diagrams	1.7
		Design Critics	13.3
		Logging	1.3
		Total	20.0

considered required annotations to 22.6 percent of Prevayler's source code, 12.1 percent of JFreeChart's code, and 20.0 percent of ArgoUML's code.

3. We have applied the extraction process described in Section 5 to annotate the same features considered in the manual extractions. In the case of Prevayler and JFreeChart, the extraction was assigned to an author of the paper different from the one who performed the manual extraction.
4. To answer our first three research questions, we have collected the following data on the extraction of each system: a) the number of iterations, b) the number of semi-automatic expansions request by the algorithm, and c) the number of default actions accepted by the developers in charge of the extraction.
5. To answer the fourth research question, we measured the precision and recall of our approach after completing the extraction process for each system, considering the manual extraction as a baseline. The purpose of precision is to measure whether the proposed approach has been able to annotate relevant code (i.e., code marked in the manual extraction). Conversely, recall measures whether our approach has been able to cover all the code related to the feature's implementation. We have calculated precision and recall in the following way:

$$precision = \frac{M \cap A}{A},$$
$$recall = \frac{M \cap A}{M},$$

where $M \cap A$ denotes the number of bytes annotated in both extractions, A denotes the number of bytes annotated in the semi-automatic extraction, and M denotes the number of bytes annotated in the manual extraction.

In the rest of this section, we present the results achieved by our approach. For each system, the presentation of the results is organized into the following sections: extraction process (where we provide answers to our first research question), semi-automatic expansions (where we discuss the findings on our second and third research questions), and precision/recall (where we address the fourth research question).

6.3 Prevayler

6.3.1 Extraction Process

After browsing Prevayler's API, the author in charge of the extraction process initially selected the following packages as seeds:

- Monitor: package `org.prevayler.foundation.monitor`.
- Censorship: package `org.prevayler.implementation.publishing.censorship`.
- Replication: package `org.prevayler.implementation.replication`.
- Snapshot: package `org.prevayler.implementation.snapshot`.
- GZip: package `org.prevayler.foundation.gzip`.

By executing the demos included in the system distribution, the developer verified for himself that the annotations have removed the mentioned features from the system. In particular, logging messages and snapshot updates were no longer generated. Furthermore, the code fragments that request a rollback have been marked. Finally, Prevayler has a demo that relies on the *Replication* feature to synchronize the database from a client application (called `MainReplica`) with the database from a server application. By executing a projection of this demo without *Replication*, the developer verified that the synchronization service had been suspended. In particular, in the console of the `MainReplica` the following message has been issued: *Trying to find server on localhost*. The next message normally presented in this case—reporting a successful connection with the server—was not issued. Therefore, after this first iteration, the developer considered the extraction process finished.

6.3.2 Precision/Recall

Table 7 presents the total number of bytes annotated in the manual and automatic extractions. As can be observed, our approach achieved 100 percent precision for all five features. Regarding the recall, our approach achieved 100 percent coverage for *Monitor*, *Censorship*, and *GZip*. For *Replication*, the recall was 0.94. In this case, the nonmarked code included four fields from the `PrevaylerFactory` class and their respective accessor and setter methods. These fields store the IP address and port number of the server application. Because the demo considered in the study assumes the server is on the localhost, such fields were not accessed during its execution.

For *Snapshot*, the recall was 59 percent. In this case, the nonmarked code included the members of the classes `PrevaylerFactory` and `PrevaylerDirectory`, which are responsible for configuring snapshot parameters (including file directory and name, serialization strategy, and so on). Essentially, the nonmarked code can be considered a client of the snapshot main code. For this reason, it was not annotated.

TABLE 7
Prevayler Results (M = Manual Extraction; A = Annotation Algorithm)

Feature	Bytes			Precision	Recall
	$M - A$	$M \cap A$	$A - M$		
Monitor	0	6,725	0	1	1
Censorship	0	4,393	0	1	1
Replication	715	11,175	0	1	0.94
Snapshot	4,628	6,880	0	1	0.59
GZip	0	3,866	0	1	1

6.4 JFreeChart

6.4.1 Extraction Process

After analyzing JFreeChart's API, the author in charge of the extraction observed that the program elements related to *Pie Charts* have the substring `Pie` in their names. Therefore, he decided to use the regular expression `*Pie*` as the seed for this feature. In a similar way, the regular expression *3D* was selected for the feature *3D Charts*. Using such seeds, it was possible to extract both features in a single iteration.

Unlike the previous features, the developer in charge of the extraction did not detect any naming pattern related to the feature *Statistical Charts*. Therefore, he decided to use the package `org.jfree.data.statistics` as the initial seed for this feature. However, the execution of the algorithm with this seed resulted in 430 semi-automatic expansions. By analyzing such expansions, the developer found that most were from six classes, which have a prefix `Statistical` (e.g., class `StatisticalBarRenderer`) or `BoxAndWhisker` (e.g., class `BoxAndWhiskerToolTipGenerator`). By browsing the JFreeChart's API, he found that these classes are also responsible for the implementation of statistical charts. Therefore, he decided to execute the algorithm again, extending the previous seed with the regular expressions `*Statistical*` and `*BoxAndWhisker*`. In this second execution, only 39 semi-automatic expansions were raised (in classes not directly related to *Statistical Charts*). Therefore, the developer considered the extraction process complete.

To test the extracted SPL, the developer relied on a set of client programs, including demos distributed with JFreeChart. The demos that plot *Pie Charts*, *3D Charts*, or *Statistical Charts* stopped compiling when a projection of the framework was generated with the mentioned features disabled.

TABLE 8
Semi-Automatic Expansions for JFreeChart

Rules	Pie	3D	Stats
SE1	10	2	18
SE2	0	0	0
SE3	8	4	4
SE4	3	1	13
SE5	0	0	4
Total	21	7	39

6.4.2 Semi-Automatic Expansions

Table 8 presents the number of semi-automatic expansions reported during the described extraction process. For the features *Pie Charts* and *3D Charts*, 21 and seven semi-automatic expansions were reported, respectively. For *Statistical Charts*, 39 semi-automatic expansions required authorization. For all three features, all the default actions suggested by the algorithm were accepted.

6.4.3 Precision/Recall

Table 9 presents the number of bytes annotated in the manual and automatic extractions. As in the Prevayler experiment, our approach achieved 100 percent precision. The recall was 0.99 for all three features; in a small number of situations, the algorithm was not able to reach the manually marked code. As illustrated in Fig. 9, for *Pie Charts*, for example, the algorithm was able to annotate all the statements of a method called `isEmptyOrNull`, which verifies if a particular set of pie charts is empty or null, except for a single statement that returns `true`. In a similar way, for *Statistical Charts*, it was able to annotate all the statements of a method called `drawItem`, which plots items of a particular type of statistical data set, except the first statements that test whether this data set is visible on the display. The developer in charge of the manual extraction annotated the whole method bodies in both examples.

6.5 ArgoUML

6.5.1 Extraction Process

Table 10 describes the feature seeds considered in the ArgoUML case study. Essentially, the developer has selected as seeds the whole packages responsible for the implementation of the feature. Finding such packages was not difficult because their names usually include the feature's name. For example, the packages selected for design critics have the substring `cognitive` in their names.[9] In most cases, a second iteration was required to include in the seeds a UI's listener class (e.g., `org.argouml.uml.ui.ActionStateDiagram`).

In this third study, the feature *ActivityDiagram* depends on *StateDiagram*. In fact, activity diagrams are usually described as a specialized form of state diagrams. In the ArgoUML implementation, this dependency is manifested

9. ArgoUML's developers claim that design critics is an attempt to support UML designers in cognitive tasks, including decision-making, decision ordering, and task-specific design understanding [37].

TABLE 9
JFreeChart Results (M = Manual Extraction; A = Annotation Algorithm)

Feature	Bytes			Precision	Recall
	$M - A$	$M \cap A$	$A - M$		
Pie Charts	1,005	383,165	0	1	0.99
3D Charts	382	172,444	0	1	0.99
Statistical Charts	2,904	348,654	0	1	0.99

by many classes responsible for activity diagrams inheriting from classes related to state diagrams.

6.5.2 Semiautomatic Expansions

Table 12 presents the number of semi-automatic expansions (SE) reported during the ArgoUML extraction process. Two observations should be made about the presented values. First, the number of SEs is significant, at least in absolute terms (295 SEs, almost half of them associated with *Design Critics*). However, such numbers represent, on average, one SE reported for each 6.18 KB of source code annotated in the extraction. We consider this average fully acceptable, when compared with the amount of work required to annotate the same features manually.

Second, the developer did not accept the default actions associated with the proposed semi-automatic expansions in only three cases. For example, the default action associated with an expansion SE4 was not accepted for the following code:

```
1: cls= org.apache.log4j.Logger.class;
2: ....
3: cls= Class.forName("... api.MDRManager");
```

In this example, a projection of the system without the marked code (line 1) will result in a syntax error. On the other hand, the default action for SE4 is "annotate the left-hand side and its references." However, in this particular case, it is not possible to authorize the propagation of the annotation because the local variable `cls` is reused to store other `Class` values in the subsequent assignments (as in line 3). For this reason, the developer did not accept the default action suggested by our tool. Instead, he manually marked only the assignment in line 1.

```
1: boolean isEmptyOrNull(PieDataset dataset) {
2:     if (dataset == null) {
3:       return true;
4:     }
5:     int itemCount= dataset.getItemCount();
6:     if (itemCount == 0) {
7:       return true;
8:     }
9:     for (...) {
10:      ...
11:    }
12:    return true;   // non-marked statement
13: }
```

Fig. 9. Example of incomplete method body annotation in JFreeChart.

6.5.3 Precision/Recall

Table 11 presents the total number of bytes annotated in the manual and automatic extractions. For *Activity Diagrams*, *Design Critics*, and *Logging*, both the precision and the recall exceeded 89 percent, which we consider a positive result for a semi-automatic approach. Essentially, the recall was not 100 percent due to limitations of the defined seeds in reaching all parts of the manual marked code. For example, the seeds chosen for *Design Critics* were not able to mark a class that implements a parser that reads a ToDo list from a XML file (because in principle this parser is able to deal with any XML document). On the other hand, the precision was not 100 percent because in many parts of the code the developer in charge of the manual extraction did not expand an annotation to its enclosing context. For example, in many situations, he annotated the whole body of a method, but did not expand the annotation to the method's declaration and calls. To reach a consensus, we discussed the detected divergences with this developer, and he has agreed that a broader annotation would be the recommended solution in such cases. In summary, the precision was not 100 percent due to nonoptimal annotations carried in the manual extraction, which is a normal event in any repetitive process conducted by humans.

For *State Diagrams*, the recall was 88 percent due to the defined seed's inability to reach some parts of the feature's code. The precision was 87 percent, but for a different reason. Because *ActivityDiagram* depends on *StateDiagram*, our algorithm has propagated *StateDiagram*'s color to program elements related to *ActivityDiagram*. For example, the class `ActivityDiagramRenderer` has been marked because it is a subclass of `StateDiagramRenderer`. The justification for this propagation is that our algorithm assumes that it is not possible to derive a product with *StateDiagram* disabled, but with *ActivityDiagram* enabled. However, by using logical operators in `#ifdef` expressions, the developer in charge of the manual extraction figured out a strategy to enable such product configurations. First, he did not expand *StateDiagram*'s color to

TABLE 10
Feature Seeds and Number of Iterations in the ArgoUML Case Study

Feature	Seeds	# Iter.
State	Two packages and two classes	2
Activity	Three packages and one class	1
Critics	Ten packages and two classes	2
Logging	One package	2

TABLE 11
ArgoUML Results (M = Manual Extraction; A = Annotation Algorithm)

Feature	KB			Precision	Recall
	$M - A$	$M \cap A$	$A - M$		
State Diagram	38.4	290.8	42.1	0.87	0.88
Activity Diagram	6.3	142.3	9.4	0.94	0.96
Design Critics	54.5	1,211.7	8.8	0.99	0.96
Logging	3.7	106.8	12.6	0.89	0.97

classes like `ActivityDiagramRenderer`, which explains the existence of 42.1 KB of source code annotated only by our algorithm (and therefore the 87 percent value for precision). Second, he used logical expressions such as `STATE || ACTIVITY` in the `#ifdefs` that delimit classes like `StateDiagramRenderer`. In this way, he could guarantee that such classes will be included in products where `STATE=FALSE`, but `ACTIVITY=TRUE`.

6.6 Discussion

Our annotation approach has contributed to reduce the time and effort required for the extraction of the features considered in the case studies. For example, the developer in charge of the manual extraction dedicated an entire month to understanding ArgoUML's design and implementation and to annotate the code with `#ifdefs` (working on average eight hours per day). Furthermore, an extra five days were dedicated to testing and fixing errors in the pruned code. On the other hand, using the semi-automatic approach, we were able to extract a product line for ArgoUML in five days, including two days for understanding its implementation and three days for applying the proposed extraction process (including the definition of the seeds, the mapping of feature dependencies, the execution of the algorithm, and the evaluation of semi-automatic expansions). It is also important to highlight that the semi-automatic extraction was not followed by a testing phase. Instead of testing the annotated code, we have compared it with the manually marked code, to calculate the precision and recall results previously presented on this section. However, we can estimate that the effort required for testing the product line would be similar to that required in the case of manual extraction.

Despite such encouraging results, the proposed approach presumes reasonable knowledge of the structure of the target system and particularly about the implementation of the features under extraction. Essentially, this knowledge is required to define the feature seeds, authorize semi-automatic expansions, detect subtle feature interactions, and systematically test the SPL core and products. In other words, we should not assume that developers will be able to execute the annotation algorithm without a minimal knowledge of the internals of the target program. In fact, we envision the proposed extraction process being adopted by the developers of the non-SPL-based version of the target system.[10]

In the remainder of this section, we summarize other lessons learned through our case studies:

- The selection of the seeds is critical to the success of the proposed algorithm. Ideally, it should be possible to reach any code associated with the features under extraction from the defined seeds. For this reason, natural candidates for seeds are packages fully dedicated to the implementation of the features. For example, for *Design Critics* in the ArgoUML study, we selected 10 packages with the substring `cognitive` in their names. By contrast, it is more challenging to apply the proposed approach when the feature code is not confined to a small number of related packages or when the feature code does not follow a particular naming convention. For example, the classes responsible for *Statistical Charts* in JFreeChart are distributed over three nonrelated packages (`data.statistics`, `chart.labels`, and `chart.renderer`). Initially, the developer provided only the first package as seed. He was only able to detect the need to include the remaining packages due to the high number of semi-automatic expansions triggered in the first execution of the algorithm. If the developer had finished the extraction in the first execution, the recall rate would have been of 71.5 percent (instead of the reported 99 percent rate achieved after refining the seeds to include the classes in the other packages).
- The default expansions suggested to handle semi-automatic expansions have proved very useful. For example, in the ArgoUML example, 292 out of 295 default expansions have been accepted. We explain the large number of default expansions accepted as follows: First, for SE1 to SE4, refusal scenarios usually occurred when multiple colors were tangled in the same program abstractions. For example, the rejection of the default expansion associated with SE1 means that two or more colors are tangled in an

TABLE 12
Semi-Automatic Expansions for ArgoUML (Nonaccepted Default Actions Are Indicated within Parentheses)

Rules	State	Activity	Critics	Log	Total
SE1	20(1)	18(1)	33	2	73
SE2	0	0	0	0	0
SE3	23	7	35	1	66
SE4	49	14	44	1(1)	108
SE5	8	1	24	15	48
Total	100	40	136	19	295

10. Therefore, in this discussion, we refer to the software developers in charge of extracting an SPL simply as developers.

expression. Similarly, failure to accept SE4's default expansion may be explained by the reuse of a local variable to store values generated by code associated with more than one feature. In general, these situations involve code that is more difficult to understand and evolve (or even programming practices that should be avoided, such as the reuse of variables). Therefore, we could explain the massive number of authorized default actions for SE1-SE4 by the fact that developers—at least the developers of the three evaluated systems—have a tendency to avoid the (re)use of program abstractions (e.g., expressions, functions, local variables, etc.) to denote values associated with different features. Finally, for the default expansions associated with SE5, the explanation is different. In this particular case, default expansions are usually accepted because it is a bad programming practice to have expressions with subtle side effects in loop or conditional statements.

- As mentioned in Section 3, our approach delegates to developers the following tasks:
  - to make sure that the features under extraction are optional (i.e., that removing their code will not affect the core), and
  - that dependencies between optional features are managed externally by developers.

  For example, in the ArgoUML study, *ActivityDiagram* depends on *StateDiagram*. Our approach assumes that this dependency should be discovered and managed by SPL developers. In particular, developers should ensure that products will never be delivered with *ActivityDiagram* enabled and *StateDiagram* disabled. However, it is not impossible to obtain such a product. In fact, the developer in charge of the manual extraction was able to generate such a product. However, for this purpose, he had to define Boolean expressions for some `#ifdef` directives. Nevertheless, our solution and, in fact, the CIDE tool are not able to associate logical expressions to blocks of code implementing optional features.
- Annotation-based and composition-based approaches are not mutually exclusive. In particular, composition-based mechanisms can provide the well-known benefits of modular programming (such as parallel development, separate compilation, reusability, etc.). We also claim that sophisticated composition-based mechanisms—including aspects—have their uses, particularly in handling homogeneous crosscutting concerns such as profiling and transactions [44]. Finally, even when aspects are recommended, our annotation algorithm can help to identify candidate code for aspect-oriented refactorings.
- The proposed extraction process aims at the extraction of a first SPL-based implementation of an existing system. Therefore, after this first SPL has been extracted, developers should maintain and evolve the annotations accordingly. In the case of changes within the limits of an annotated code fragment, CIDE automatically annotates the changed or inserted elements. However, in the case of changes affecting nonmarked parts of the code (e.g., inserting a new call to a method associated with a given feature in a nonmarked part of the code), developers should manually annotate the new program elements. The same happens to code evolved due to refactoring operations. To better handle evolution issues, we plan to extend our tool with a form of "continuous annotations," i.e., whenever evolved code matches the defined seeds it will be automatically annotated.

6.7 Threats to Validity

Four main factors can affect the validity of the results presented in this paper. First, our three subject programs might not be representative of all product line extraction scenarios. We are aware of this threat and to minimize it we decided to evaluate two well-known and complex open-source systems (JFreeChart and ArgoUML) and a system that has been previously used in other studies of SPL extraction (Prevayler). The second threat is related to the validity of the extracted features. For example, it is possible that we have accidentally selected features that fit the requirements of our approach better (or worse). To mitigate this threat, we decided to select two kinds of features: functional features related to the core domain of each target system (e.g., charts in JFreeChart and diagrams in ArgoUML) or typical nonfunctional concerns (e.g., logging in the Prevayler and ArgoUML).

As a third threat, in the Prevayler and JFreeChart studies, the manual extraction has been conducted by one of the paper's authors (i.e., a developer with full knowledge of our approach and its limitations). To minimize this possible threat, we conducted a third-case study, involving larger and more complex features than in the previous systems. More importantly, in this third study, our semi-automatic approach was contrasted with a manual extraction conducted independently by a software developer with no knowledge about our work.

Finally, our approach depends on feedback and input provided by the target system developers. For example, developers should define the feature seeds and map feature dependencies. Manual effort is also required to test product specific code and to approve semi-automatic expansions.

7 Related Work

Work related to our research can be divided into five main groups: experience reports, tool support, type systems, feature interaction, program understanding, and program slicing.

7.1 Experience Reports

Lopez-Herrejon et al. have evaluated the modularization of features using five technologies: AspectJ, Hyper/J, Jiazzi, Scala, and AHEAD [32]. As a case study, they used a simple expression-product line, including variabilities such as data types and operations. Because it is very simple, the implementation of this SPL did not require fine-grained extensions as do those that are supported only by annotation-based approaches, such as preprocessors and CIDE.

Kästner et al. have documented the extraction of features from a database management system (Berkley DB) using

aspect-oriented programming (AspectJ) [23]. Because the Berkley DB is an industrial-sized and complex system, they faced many problems in the extraction, mainly due to AspectJ's limitations to handling fine-grained variabilities. For example, features that require extensions in nested statements cannot be directly extracted to aspects. In such situations, they had to insert hook methods in the code just to enable join points that can be instrumented by AspectJ. The need to rely on hook methods when using compositional technologies—such as AspectJ—has also been reported by Murphy et al. in another study on the extraction of features [33]. Finally, the same limitations have been cited by Adams et al. in a study that refactored conditional compilation to aspects [1].

Godil and Jacobsen have previously used aspect-oriented programming to extract a set of features from Prevayler [16]. Although not detailed in the work, their modularization presents the same problems reported by Kästner et al. for the Berkley DB study, including complex and fragile pointcuts and obfuscated code due to hook methods.

Bruntink et al. describe an in-depth study involving the characterization of the tracing concern in a legacy system with about 15 million lines of C code [7]. Although tracing is usually considered a "simple concern," they pointed out that this industrial-sized implementation presents remarkable variabilities. The observed variabilities have been classified as accidental (e.g., improper use of idioms) or essential (e.g., particular needs of a subsystem). Furthermore, they state that the extraction of accidental variabilities to aspects can lead to implementations with a small degree of quantification (*one-aspect-per-function* implementations). However, this problem does not exist if annotation-based solutions are used as, in this case, the modularization is virtual (i.e., there is no physical extraction of crosscutting concerns to separate modules). For this reason, SPL extraction methods using annotation technologies is more suited to automation.

7.2 Tool Support

Liu et al. describe a formal approach to feature-oriented refactoring [31]. The proposed approach, which is based on the composition language AHEAD [3], has been used to extract features from the Prevayler persistence framework. Despite the fact that AHEAD only supports coarse-grained method extensions, the authors do not detail the enabling transformations that were required in the base program to extract features to refinements (the modularization units supported by AHEAD). They only mention that statement reorderings have been required in some situations.

Binkley et al. developed the AOP-Migrator tool, an Eclipse plug-in that automates six refactorings commonly used to support migration from OOP to AOP [5], [6]. However, when the object-oriented code does not fulfill the preconditions assumed by the supported refactorings, developers should manually transform the code to enable the refactorings. These transformations, called object-oriented transformations, typically include statement reorderings (e.g., moving a statement to the beginning of a method's body) or method extraction (to enable a join point). FLiP is another tool for extracting product variations from Java classes into AspectJ aspects [40]. Therefore, it has the same limitations we have previously observed for the AOP-Migrator tool.

Change-Oriented Programming (ChOP) is a FOP-based programming approach in which features are represented by a set of changes applied to a base system [13], [12]. As in the case of CIDE-based product lines, ChOP requires the instrumentation of an IDE, but this time to generate editing logs. Therefore, the main difference between our approach and ChOP is that our annotation algorithm does not depend on any previous information about the evolution of the target system, whereas ChOP requires a complete record generated by an instrumented IDE.

In previous work, we have demonstrated the importance of object-oriented transformations using four medium-sized systems as case studies [35], [34]. We have also designed a tool, called TransformationMapper, to guide developers in the application of object-oriented transformations. On the other hand, by relying on an annotative approach, the feature-extraction algorithm described in this paper does not impose any preconditions to the base code. In other words, object-oriented transformations are not needed, which has been crucial to achieving the degrees of automation observed in our case studies. Furthermore, our approach does not require the implementation of "infrastructure code," such as pointcut descriptors and advice signatures. Finally, in our approach, annotations can be performed one feature at a time, without affecting the features already annotated or the system's core.

7.3 Type Systems

Kästner and Apel formalized a type system that guarantees that all variants from an SPL are well typed [22]. For example, their type system can be used to detect situations such as dangling method calls (i.e., methods removed in a variant, but that are still called). Not surprisingly, the proposed type rules—defined for an extension of Feather-Weight Java [20]—resemble the color propagation phase from our annotation algorithm. In fact, the purpose of color propagation is precisely to remove all references to the defined seeds from the codebase, thereby eliminating the risk of variants carrying dangling references. Other approaches to check that all programs in an SPL are type safe have been proposed by Thaker et al. [43] and Czarnecki and Pietroszek [11].

7.4 Feature Interaction

The implementation of one feature may interfere with other features, which is a recurrent problem in many domains, such as in telecommunication systems. For this reason, several techniques have been proposed to detect feature interactions. Such techniques can generally be classified as offline (i.e., techniques applied at design time) or online (i.e., techniques applied while the features are actually running) [29]. Offline techniques concentrate on the use of formal methods to both validate benign interactions and detect unpredicted ones. When formal methods are applied, unexpected interactions usually lead to the inconsistency or unsatisfiability of the (logical) formulas describing the core service and its features [8]. On the other hand, online techniques usually advocate the existence of a central component (a feature manager) to regulate the activation

and deactivation of features at runtime. As mentioned on Section 3, we regard feature interactions as an orthogonal problem to our annotation approach. Essentially, our goal is to mark code in order to bootstrap existing products into an SPL. In particular, our assumption is that the given feature seeds denote optional components in the product under extraction. Therefore, removing such components must not have an unexpected effect on the core semantics.

7.5 Program Understanding Tools

Several approaches have been proposed to help developers and maintainers to manage concerns and features. For example, concern graphs model the subset of a software system associated with a specific concern [38], [39]. The main purpose is to provide developers with an abstract view of the program fragments related to a concern. FEAT is a tool that supports the concern graph approach by enabling developers to build concern graphs interactively, as result of program investigation tasks. Aspect Browser [17] and JQuery [21] are other tools that rely on lexical or logic queries to find and document code fragments belonging to a certain concern (or feature). Like CIDE, these program investigation tools are based on logical views of the source code (i.e., they do not provide a means to separate concerns in physical modules). On the other hand, they cannot be used to annotate and to generate projections of the source code without the code for a given feature, which is crucial when extracting SPLs.

7.6 Program Slicing

Slicing is a technique for simplifying programs, typically by computing all statements that may affect the value of a variable at a specific program location (and removing the remaining statements) [46], [19]. Therefore, slicing may help with program comprehension because it preserves the semantics of the original program while reducing its size. However, slices may include much more information than is reasonable in a software maintenance task. This is particularly the case in highly cohesive programs in which the value of a variable may affect many other variables. For example, Harman et al. have recently demonstrated the prevalence of large dependence clusters (i.e., mutually interdependent statements) in C programs, which has a clear impact on program comprehension [18].

To summarize, most recent research on SPL implementation has been centered on composition-based approaches, mainly AOP and FOP. On the other hand, there are few reports on the design of tools to avoid the problems associated with annotation-based product lines. The recently proposed CIDE tool is an exception to this general rule. Therefore, in this paper, we have investigated an algorithm for feature extraction based on disciplined and visual annotations, as supported by the CIDE tool.

8 Conclusions

In this paper, we have made the following contributions:

- We proposed a semi-automatic approach to annotate variabilities in SPLs. Our approach relies on the CIDE tool to annotate the lines of code in charge of implementing the variabilities from a given SPL. Initially, developers must provide a set of feature seeds and a color. The proposed approach propagates this color by the program locations that reference the provided seeds in a direct or indirect way. Whenever possible and safe, it also expands the color by the lexical context of the elements already annotated. When expansions are required but their safety cannot be easily inferred, our approach suggests default expansions to be approved by developers.
- We successfully applied the proposed algorithm to three nontrivial systems. In all cases, our algorithm achieved recall and precision rates ranging from 59 to 100 percent. The case studies have demonstrated the importance of the feature seeds selection and the importance of having client programs that exercise all the services provided by the features. In particular, such client programs are critical for detecting unmarked code and therefore to trigger a new iteration in the extraction process.
- We argued that the proposed algorithm provides degrees of automation well beyond the existing tools for feature extraction. The main reason is that such tools are based on compositional languages, such as AspectJ and Jak. Therefore, they are directly affected by the coarse-grained model of program extensions supported by such languages [24]. To make the object-oriented code conform to this model, compositional approaches require the transformation of the code in a way that cannot be easily automated by any tool [35], [34]. To the best of our knowledge, the proposed solution is the first algorithm for feature extraction that relies on an annotative approach.

The proposed approach is not recommended for optional features whose implementation heavily affects user-interface concerns. In such cases, removing the feature may compromise the layout and usability of the interface. It is also not recommended for features coupled to external resources, such as databases. Furthermore, the current implementation of the proposed annotation algorithm does not handle features associated with the following Java elements: exceptions, enums, and generic types. We have plans to support such elements in further versions.

In the near future, we also have plans to apply the proposed algorithm to other case studies. Finally, our current implementation is limited to the pure-Java based version of CIDE. Therefore, we have plans to extend and adapt it to other languages.

Our tool is available for downloading from: http://www.dcc.ufmg.br/~mtov/cideplus.

Acknowledgments

This research has been supported by grants from FAPEMIG, CAPES, and CNPq. The authors thank Christian Kästner and Sven Apel for their feedback on an earlier draft of this paper and Rogel Garcia for helping them with the algorithm implementation. They also thank the anonymous referees and the associated editor for valuable insights and comments on earlier versions of this paper.

REFERENCES

[1] B. Adams, W.D. Meuter, H. Tromp, and A.E. Hassan, "Can We Refactor Conditional Compilation into Aspects?" *Proc. Eighth ACM Int'l Conf. Aspect-Oriented Software Development,* pp. 243-254, 2009.

[2] M. Antkiewicz and K. Czarnecki, "FeaturePlugin: Feature Modeling Plug-In for Eclipse," *Proc. OOPSLA Workshop Eclipse Technology eXchange,* pp. 67-72, 2004.

[3] D. Batory, "Feature-Oriented Programming and the AHEAD Tool Suite," *Proc. 26th Int'l Conf. Software Eng.,* pp. 702-703, 2004.

[4] D. Beuche, "Modeling and Building Software Product Lines with Pure::Variants," *Proc. 11th Int'l Conf. Software Product Lines,* pp. 143-144, 2007.

[5] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Automated Refactoring of Object Oriented Code into Aspects," *Proc. IEEE 21st Int'l Conf. Software Maintenance,* pp. 27-36, 2005.

[6] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella, "Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects," *IEEE Trans. Software Eng.,* vol. 32, no. 9, pp. 698-717, Sept. 2006.

[7] M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwe, "Simple Crosscutting Concerns Are Not So Simple," *Proc. Sixth Int'l Conf. Aspect-Oriented Software Development,* pp. 199-211, 2007.

[8] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Computer Networks,* vol. 41, no. 1, pp. 115-141, 2003.

[9] P. Clements and L.M. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[10] M.V. Couto, M.T. Valente, and E. Figueiredo, "Extracting Software Product Lines: A Case Study Using Conditional Compilation," *Proc. 15th European Conf. Software Maintenance and Reeng.,* pp. 191-200, 2011.

[11] K. Czarnecki and K. Pietroszek, "Verifying Feature-Based Model Templates against Well-Formedness OCL Constraints," *Proc. Fifth Int'l Conf. Generative Programming and Component Eng.,* pp. 211-220, 2006.

[12] P. Ebraert, A. Classen, P. Heymans, and T. D'Hondt, "Feature Diagrams for Change-Oriented Programming," *Proc. 10th Int'l Conf. Feature Interactions,* pp. 107-122, 2009.

[13] P. Ebraert, J. Vallejos, P. Costanza, E.V. Paesschen, and T. D'Hondt, "Change-Oriented Software Engineering," *Proc. Int'l Conf. Dynamic Languages,* pp. 3-24, 2007.

[14] M.D. Ernst, G.J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Trans. Software Eng.,* vol. 28, no. 12, pp. 1146-1170, Dec. 2002.

[15] W.B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Trans. Software Eng.,* vol. 31, no. 7, pp. 529-536, July 2005.

[16] I. Godil and H.-A. Jacobsen, "Horizontal Decomposition of Prevayler," *Proc. 15th Conf. Centre for Advanced Studies on Collaborative Research,* pp. 83-100, 2005.

[17] W.G. Griswold, J.J. Yuan, and Y. Kato, "Exploiting the Map Metaphor in a Tool for Software Evolution," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 265-274, 2001.

[18] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence Clusters in Source Code," *ACM Trans. Programming Languages and Systems,* vol. 32, no. 1, pp. 1-33, 2009.

[19] M. Harman and R. Hierons, "An Overview of Program Slicing," *Software Focus,* vol. 2, no. 3, pp. 85-92, 2001.

[20] A. Igarashi, B.C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," *ACM Trans. Programming Languages and Systems,* vol. 23, no. 3, pp. 396-450, 2001.

[21] D. Janzen and K.D. Volder, "Navigating and Querying Code without Getting Lost," *Proc. Second Int'l Conf. Aspect-Oriented Software Development,* pp. 178-187, 2003.

[22] C. Kästner and S. Apel, "Type-Checking Software Product Lines a Formal Approach," *Proc. 23rd Int'l Conf. Automated Software Eng.,* pp. 258-267, 2008.

[23] C. Kästne, S. Apel, and D. Batory, "A Case Study Implementing Features Using AspectJ," *Proc. 11th Int'l Software Product Line Conf.,* pp. 223-232, 2007.

[24] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," *Proc. 30th Int'l Conf. Software Eng.,* pp. 311-320, 2008.

[25] C. Kästner, S. Apel, S.S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake, "On the Impact of the Optional Feature Problem: Analysis and Case Studies," *Proc. 13th Int'l Software Product Line Conf.,* pp. 181-190, 2009.

[26] C. Kästner, S. Trujillo, and S. Apel, "Visualizing Software Product Line Variabilities in Source Code," *Proc. Second Int'l Workshop Visualisation in Software Product Line Eng.,* pp. 303-313, 2008.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, "An Overview of AspectJ," *Proc. 15th European Conf. Object-Oriented Programming,* pp. 327-355, 2001.

[28] C.W. Krueger, "Easing the Transition to Software Mass Customization," *Proc. Fourth Int'l Workshop Software Product-Family Eng.,* pp. 282-293, 2001.

[29] A.F. Layouni, L. Logrippo, and K.J. Turner, "Conflict Detection in Call Control Using First-Order Logic Model Checking," *Proc. Ninth Int'l Conf. Feature Interactions in Software and Comm. Systems,* pp. 66-82, 2007.

[30] G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok, "How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification," *Science of Computer Programming,* vol. 55, nos. 1-3, pp. 185-208, 2005.

[31] J. Liu, D. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications," *Proc. 28th Int'l Conf. Software Eng.,* pp. 112-121, 2006.

[32] R. Lopez-Herrejon, D. Batory, and W.R. Cook, "Evaluating Support for Features in Advanced Modularization Technologies," *Proc. 19th European Conf. Object-Oriented Programming,* pp. 169-194, 2005.

[33] G.C. Murphy, A. Lai, R.J. Walker, and M.P. Robillard, "Separating Features in Source Code: An Exploratory Study," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 275-284, 2001.

[34] M. Nassau, S. Oliveira, and M.T. Valente, "Guidelines for Enabling the Extraction of Aspects from Existing Object-Oriented Code," *J. Object Technology,* vol. 8, no. 3, pp. 1-19, 2009.

[35] M. Nassau and M.T. Valente, "Object-Oriented Transformations for Extracting Aspects," *Information and Software Technology,* vol. 51, no. 1, pp. 138-149, 2009.

[36] D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.,* vol. 2, no. 1, pp. 1-9, Mar. 1976.

[37] J.E. Robbins and D.F. Redmiles, "Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML," *Information and Software Technology,* vol. 42, no. 2, pp. 79-89, 2000.

[38] M.P. Robillard and G.C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," *Proc. 24th Int'l Conf. Software Eng.,* pp. 406-416, 2002.

[39] M.P. Robillard and G.C. Murphy, "Representing Concerns in Source Code," *ACM Trans. Software Eng. and Methodology,* vol. 16, no. 1, pp. 1-38, 2007.

[40] S. Soares, F. Calheiros, V. Nepomuceno, A. Menezes, P. Borba, and V. Alves, "Supporting Software Product Lines Development: FLiP —Product Line Derivation Tool," *Proc. 23rd Conf. Object-Oriented Programming, Systems, Languages, and Applications, Tool Demonstrations,* pp. 737-738, 2008.

[41] H. Spencer, "#ifdef Considered Harmful, or Portability Experience with C News," *Proc. USENIX Conf.,* pp. 185-197, 1992.

[42] V. Sugumaran, S. Park, and K.C. Kang, "Introduction to the Special Issue on Software Product Line Engineering," *Comm. ACM,* vol. 49, no. 12, pp. 28-32, 2006.

[43] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," *Proc. Sixth Int'l Conf. Generative Programming and Component Eng.,* pp. 95-104, 2007.

[44] M.T. Valente, C. Couto, J. Faria, and S. Soares, "On the Benefits of Quantification in AspectJ Systems," *J. Brazilian Computer Soc.,* vol. 16, no. 2, pp. 133-146, 2010.

[45] J. van Gurp, J. Bosch, and M. Svahnberg, "On the Notion of Variability in Software Product Lines," *Proc. IEEE/IFIP Second Working Conf. Software Architecture,* pp. 45-54, 2001.

[46] M. Weiser, "Program Slicing," *Proc. Fifth Int'l Conf. Software Eng.,* pp. 439-449, 1981.

Marco Tulio Valente received the PhD degree in computer science from the Federal University of Minas Gerais, Brazil, where he is now an assistant professor in the Computer Science Department. His research interests include software product lines, software architecture, and software maintenance and evolution. He is a member of the ACM, the IEEE Computer Society, and the Brazilian Computer Society.

Virgilio Borges received the MSc degree in computer science from the Pontifical Catholic University of Minas Gerais. He is a lecturer in the Department of Informatics at COTEMIG, Brazil. His research interests include programming languages and software product lines.

Leonardo Passos received the master's degree in computer science from the Federal University of Minas Gerais, Brazil. He is working toward the PhD degree in the Electrical and Computer Engineering at the University of Waterloo, Canada. His research interests include programming languages and software product lines.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.