JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Reverse Engineering Variability from Product Variants

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Masterstudium

Informatik

Eingereicht von:
Lukas Linsbauer, 0956251

Angefertigt am:
Institute for Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed M. Sc.

Mitwirkung:
Dr. Roberto Erick Lopez-Herrejon M. Sc.

Linz, September, 2013

Reverse Engineering Variability from Product Variants

Lukas Linsbauer
Systems Engineering and Automation
Johannes Kepler University Linz
Austria
k0956251@students.jku.at

September 26, 2013
(last updated February 17, 2014)

Kurzfassung

Unternehmen entwickeln oft ähnliche Softwareproduktvarianten die manche Teile gemeinsam haben und sich in anderen Teilen unterscheiden. Die Anzahl solcher Varianten steigt mit der Zeit durch das Kopieren von bestehenden Varianten und anschließendes Anpassen um neuen Anforderungen gerecht zu werden. Ab einem gewissen Punkt ist die Wartung von bestehenden und die Erstellung von neuen Varianten kaum mehr durchführbar. Änderungen an Features oder das Beheben von Fehlern muss für jede Variante wiederholt werden die das betroffene Feature implementiert was für eine große Anzahl von Varianten sehr fehleranfällig und kostspielig ist. Außerdem wird es mit vielen Produktvarianten sehr schwierig zu entscheiden welche Teile von welchen Varianten wiederverwendet werden sollten für die Erstellung von neuen Varianten. Eine Option wäre die bestehenden ähnlichen Softwareproduktvarianten in ein einziges, konfigurierbares System überzuführen oder gar sie von Beginn an als ein solches System zu entwickeln. Dies nimmt jedoch eine sehr lange Zeit in Anspruch und erfordert im Vorfeld hohe Investitionen von Zeit und Geld was sich Unternehmen oft nicht leisten können. Außerdem würden sie damit viel Flexibilität einbüßen in Bezug auf die Umsetzung von neuen Anforderungen und Produktvarianten für die das System ursprünglich nicht entworfen wurde.

Deshalb ist das Ziel dieser Arbeit eine teilweise Lösung für alle diese Herausforderungen zu schaffen durch automatisierte Extraktion von Variabilität aus existierenden Softwareproduktvarianten und die gezielte Wiederverwendung von bestehenden Implementierungsteilen um die Wartung zu vereinfachen und die Erstellung von neuen Varianten zu unterstützen. Zu diesem Zweck werden automatisch Traces von sowohl Features als auch Featureinteraktionen zu beliebigen Implementierungsartefakten (Quellcode, Modelle, Dokumentation, etc.) erstellt indem bestehende Produktvarianten miteinander verglichen werden. Zusätzlich werden Abhängigkeiten zwischen Implementierungsteilen extrahiert die als eine Art Variabilitätsmodell dienen können.

Abstract

Companies often develop a set of similar software product variants that have some parts in common while differing in other parts. The number of such variants often increases over time by copying existing ones and adapting them to fit new requirements. At some point the maintenance of existing variants and the creation of new ones becomes unmanagable. Changes to features or bugfixes have to be replicated in every variant that implements the feature which is error prone and costly to do for a large number of variants. Also, when creating new variants it becomes difficult to decide which assets from what existing variants to reuse. One option would be to refactor such a set of related software product variants into a single, configurable system representation or even develop them like this from the start. But this takes a very long time to do and requires a major upfront investment of time and money which companies often cannot afford. Also they then lack the flexibility when it comes to new requirements and product variants that the system was not initially designed for.

Therefore this thesis aims to provide a partial solution to all these challenges by providing automated support for reverse engineering variability from existing software product variants and guiding reuse of assets, easing their maintenance and supporting the creation of new variants. For this purpose traces from features as well as feature interactions to implementation artifacts of arbitrary types (source code, models, documentation, etc.) are established automatically by comparing existing product variants with each other. Additionally dependencies between implementation assets are extracted that can serve as some form of variability model.

Contents

Chapter 1

Introduction

Many companies develop custom software for their clients. They start out by developing an initial variant that is tailored specifically to a certain client. When others come along that need similar yet slightly differing solutions often these companies copy the closest variant they have and adapt it to fit the new clients' needs. This is repeated for every new client, which leads to an ever growing portfolio of related software product variants, a product portfolio. However, over time the number of variants will become too large to be maintained. A typical problem in maintaining large sets of related product variants are bugfixes or changes to features that have to be replicated in every product that implements this features, which becomes error prone, very inefficient and costly.

At that point companies might want to refactor their product portfolio into a single, configurable system. This would ease the maintenance significantly because bugfixes and changes to features no longer have to be made for every variant separately but instead only need to be made once. Additionally, due to the systematic reuse, time-to-market would be improved once such a system has been built. However, such a refactoring can be very costly and take a very long time (often several years) [Rubin et al., 2013] during which the company cannot just stop maintaining the existing variants or developing new variants. These processes have to run in parallel, so once the product family was refactored there is the risk that it is already outdated or will become outdated very soon, since for such systems usually the variability and all possible variants have to be planned beforehand. This is also why building such a single, configurable system from the very beginning usually is also not an option, because it is often not possible to plan for all future needs that might arise with new customers and an adaption is often difficult. Additionally building such a system requires major upfront investments in terms of time (time until the first products can hit the market is very high) and money, which companies often cannot afford.

This is why in practice many companies want to stick with their original approach of copying and adapting existing variants because it requires no major upfront investments, reduces time-to-market for the initial products and is very flexible when facing new requirements. However, they lack methodology and tool support for systematic reuse and for dealing with maintanance issues that arise with extensive duplication.

The goal of this thesis is to find a way to deal with all these issues and enabling companies to leverage the benefits of both practices. We want to allow for the high flexibility without the problems in maintenance while still not requiring major upfront investments of time and money. Also we want to combine the low time-to-market of the copying approach for the initial variants while taking advantage of the improved time-to-market for later product variants that a single, configurable system representation would allow for.

We do this by automatically reverse engineering variability either from a portfolio of already existing product variants or incrementally for every new product that is generated, allowing for easier maintenance and systematic reuse while still allowing for the flexibility of creating arbitrary new variants on demand. Even more so, using the extracted information the cloning step can be partially automated and the composition of new variants can be supported through guided reuse [Fischer, 2013]. Based on this we propose an approach for supporting the development and maintenance of a product portfolio.

Possible application scenarios for this approach are: (1) the reverse engineering of legacy software product variants into a single system representation, (2) the extension of an already existing single system representation beyond what it was initially designed for, (3) the approach being applied directly as a development paradigm (ideally from the very beginning of a software development process) to manage variability, reuse and maintenance to leverage all its benefits, or (4) simply the extraction of useful traceability and dependency information to support maintenance.

The remainder of this thesis is structured as follows: the following Chapter 2 introduces the relevant background, in Chapter 3 the basic data structures are introduced along with an example that will be used for illustration throughout this thesis. In Chapter 4 the extraction process is presented which is the core of this thesis. Chapter 5 gives a brief overview about the composition [Fischer, 2013] and Chapter 6 outlines an approach for supporting clone-and-own using the presented extraction and composition procedures. Chapter 7 gives an overview of the implementation of the extraction framework and Chapter 8 shows the results of the performed evaluation. Finally Chapter 9 talks about possible threats to validity, Chapter 10 gives an overview about related work, Chapter 11 summarizes and concludes and Chapter 12 gives an outlook on future work.

Chapter 2

Background

This chapter gives an overview about relevant background for this thesis. It introduces two concrete approaches for developing sets of related software product variants that the presented approach has to compete with.

2.1 Software Product Lines (SPLs)

Product lines have existed for a very long time in traditional manufactoring, as for example in the automotive industry. A typical example for a product line would be the manufactoring process of a car of a certain type. Cars of the same model are similar, yet not identical. They might share some common traits while varying in others. One instance of a car might have a different color, another a different engine or an extra feature like air conditioning. Each of these are features of a car that can be composed in various ways. Some features are optional and represent the variable part of the car while others may be mandatory and are the same for every instance of this type of car. This is usually expressed in a variability model (see next section). Every instance of a car of this model is then composed of a set of assets that represent certain features instead of developing each car from scratch depending on customers' needs.

This concept is also applied in software engineering in the form of strategic, planned reuse. A family of related software products is built using a set of assets, each implementing certain features. These assets are designed and developed for reuse. This way the products can be built to address certain market segments or types of customers without building every software product from scratch. Clements and Northrop define an SPL as follows:

Definition 1 *A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements and Northrop, 2002].*

Hence a software product line is an example of a single, configurable representation of a software system.

According to van d. Linden et al. [van d. Linden et al., 2007] software product line engineering comprises two life-cycles: *domain-engineering* and *application engineering* (see Figure 2.1). Domain engineering results in the common assets that together form the product line's platform. It also ensures that the platform has the variability that is needed to support the desired scope of products. Application engineering develops the products in the product line. It results in the delivered products. These two life-cycles consist of sub-processes that interact with each other. The domain engineering sub-processes result in common assets that are used in their application engineering counterparts to create products. In return the application engineering sub-processes generate feedback that is used in domain engineering to improve the common assets.

According to Clements and Northrop [Clements and Northrop, 2002] on the other hand there are three essential product line activities: *management*, *core asset development* and *product development* (see Figure 2.2), where core asset development corresponds to domain engineering and product development corresponds to application engineering [Northrop, 2008].

A product line strategy in software engineering can have a lot of benefits. It can lead to improvements in cost, quality, productivity and time-to-market [Clements and Northrop, 2002, van d. Linden et al., 2007]. However, applying the concept of product lines in software engineering efficiently and being able to handle the complexity of large product lines at all requires proper techniques and tool support. Also it requires a lot of upfront investments of time and money to initially create the product line. Also future requirements have to be known upfront in order to be able to design the system as a product line.

In addition, there exist a lot of legacy software products that may be

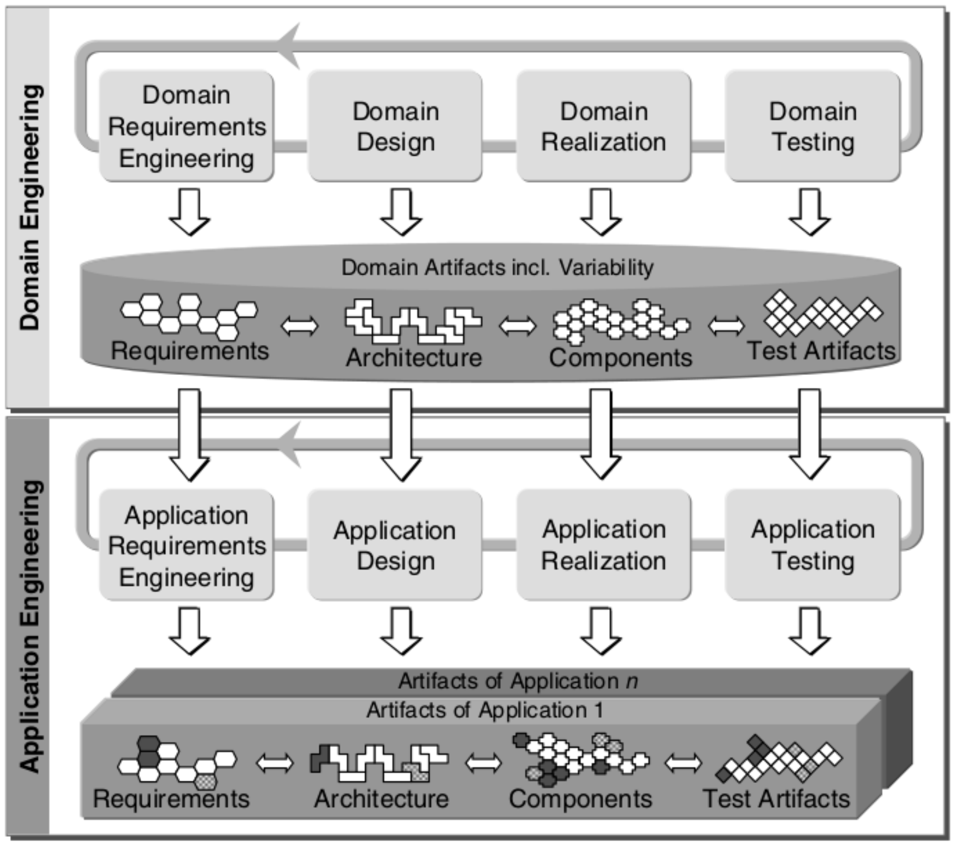


Figure 2.1: SPLE Life-Cycles [Metzger and Pohl, 2007]

similar but were not designed as a product line from the start. In such cases it may be worth it to refactor these products into a software product line in order to be able to benefit from the advantages. However, this again requires a lot of time during which the regular development process of a company cannot just stop. The process of refactoring a product portfolio into a product line and the maintenance and development of the legacy software variants have to run in parallel, which makes it even more challenging [Rubin et al., 2013].

2.2 Feature Models (FMs)

Feature Models (FMs) are used to model variability and commonality in SPLs. A *feature model* describes the features of an SPL and their relations to each other. It is a tree structure with the nodes being features. The

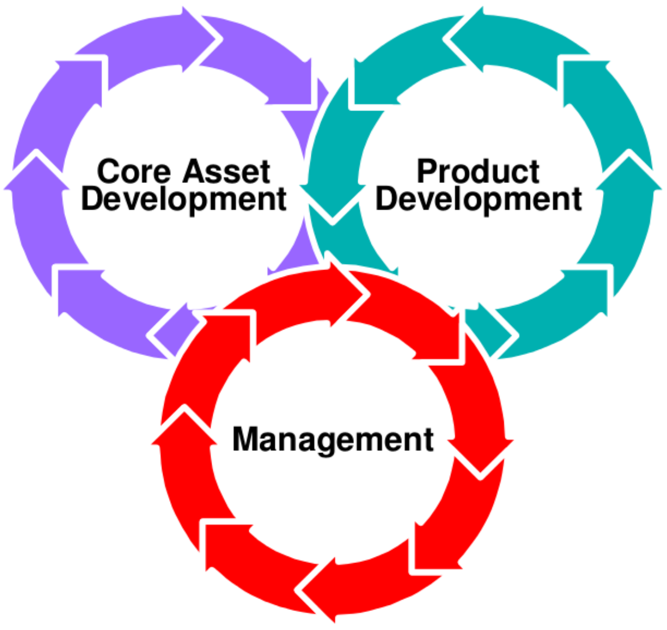


Figure 2.2: Essential Product Line Activities [Northrop, 2008]

root node of a feature model is always included in all products. A feature can only be part of a product if its parent feature is also part of it. A feature can be *mandatory* (denoted with a filled circle at the child end of an edge) or *optional* (denoted with an empty circle at the child end of an edge). A mandatory feature is part of a product whenever its parent feature is. An optional feature can but need not be part of a product if its parent is. Features can be grouped into an *inclusive-or* relation (filled arc) where one or more features of the group can be selected or an *exclusive-or* relation (empty arc) where exactly one feature must be selected. [Haslinger et al., 2011]

In addition to parent-child relations there can also be relationships between features across the tree structure of the feature model. These are called *cross-tree constriants (CTCs).* A *requires* constraint expressed that the presence of a feature `A` implies the presence of another feature `B` and is denoted as a dashed single-arrow line from `A` to `B`. An *excludes* constraint expresses that if a feature `A` is selected another feature `B` must not be selected which is denoted as a dashed double-arrow line between `A` and `B`. [Haslinger et al., 2011]

An abstract example of a feature model is given in Figure 2.3. The feature *Root* is always selected in every product. Feature *A* is mandatory and therefore also always selected. Among its three children *F*, *G* and *H* exactly one has to be selected. Among features *C* and *D* at least one has to be selected. Features *B* and *E* are both optional. However, if feature *E* is selected then also feature *H* must be selected and therefore features *F* and *G* cannot be selected. And if feature *B* is selected then *D* must not be and vice versa as they exclude each other.

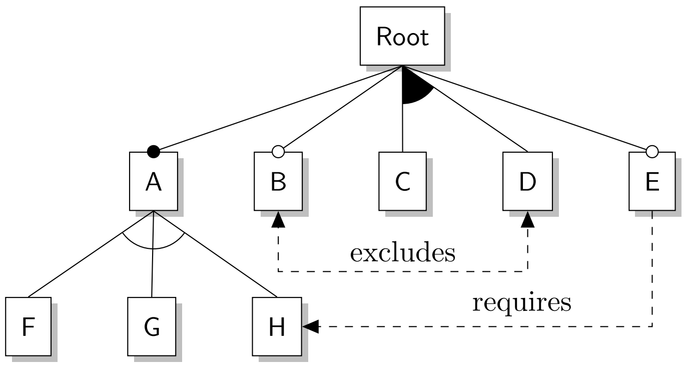


Figure 2.3: Abstract Feature Model Example

2.3 Clone-and-Own

Clone-and-Own is an ad-hoc software development paradigm that is often applied in practice, sometimes due to lack of a better alternative, sometimes because the transition to an alternative approach would be too expensive and sometimes on purpose [Dubinsky et al., 2013]. New product variants are built by cloning an existing variant that is closest to the new requirements and adapting it, possibly by also reusing parts from other existing variants in the product portfolio, to fit the new specifications.

Advantages are that this is a conceptually simple way to develop new product variants, and it is also very flexible. A new variant can be created any time without a lot of upfront investments of time and money.

However, as it lacks methodology and tool support it also comes with significant disadvantages. Due to the fact that reuse happens in an unstructured way and features are replicated in a lot of product variants simply by duplicating them the maintenance of the product portfolio becomes increasingly challenging the more variants there are. Bugfixes or changes to features have to be replicated in every product variant that implements the feature which is error prone and can become very expensive and time consuming. Also it becomes increasingly difficult for large product portfolios to decide which parts of what variants to clone when creating a new variant.

Chapter 3

Basics and Example

This chapter introduces the basic data structures along with a simple example that will be used for illustration throughout the remainder of this thesis. Consider a small portfolio of simple drawing applications of which a company has so far developed and currently maintains three variants that are shown in Table 3.1. In this domain products implement a subset of the following features: a basic drawing area (`BASE`), drawing a line on the canvas (`LINE`), drawing a rectangle on the canvas (`RECT`), wiping the canvas clean (`WIPE`) and selecting a color to draw with (`COLOR`).

Products	`BASE`	`LINE`	`RECT`	`COLOR`	`WIPE`
Product $\mathtt{P}_1$	✓	✓			✓
Product $\mathtt{P}_2$	✓	✓		✓	
Product $\mathtt{P}_3$	✓	✓	✓	✓	

Table 3.1: Initial Drawing Application Product Variants

In addition to the set of features that a variant provides we also know about each product variant its implementation artifacts, which in the case of the drawing applications are Java source code. The source code for product $\mathtt{P}_1$ is shown in Figure 3.1.

The source code will be represented in the form of a syntax tree. The chosen representation for implementation artifacts however shall be generic and general purpose. It shall not be specific to programming languages but also be applicable to e.g. models, documentation, test cases or any other kind of artifacts. The chosen representation therefore is a generic tree structure consisting of nodes with exactly one parent and an arbitrary

```
class Canvas {
  List<Line> lines;
  void wipe() {
    this.lines.clear();
  }
  ...
}
class Line {
  Line(Point start) {...}
  ...
}
class Main extends JFrame{
  initContentPane() {
    toolPanel.add(lineButton);
    toolPanel.add(wipeButton);
  }
  ...
}
```

Figure 3.1: Source Code Snippets for Product $\texttt{P}_1$ (`BASE`, `LINE`, `WIPE`)

number of children. Every node contains an artifact. This representation, since it is a very universal tree structure that is not in any way specific to a programming language, can also be used to represent arbitrary models, like UML diagrams in Ecore format.

In what follows we access elements of a tuple by means of the | operator followed by the elements's name, e.g. $T|_{Element}$ would access the element *Element* of tuple T.

Definition 2 *An* artifact $A \in \mathbb{A}$ *(given a universe of artifacts* $\mathbb{A}$*) is defined as a three-tuple* $(\texttt{Identifier}, \texttt{Uses}, \texttt{UsedBy})$ *where* `Identifier` *is an arbitrary identifier for the artifact,* $\texttt{Uses} \subseteq \mathbb{A}$ *is the set of artifacts that A uses (i.e. A depends on) and* $\texttt{UsedBy} \subseteq \mathbb{A}$ *is the set of artifacts that use A (i.e. depend on A). The following always holds:* $A_2 \in \mathbb{A} \wedge A_2 \in A|_{UsedBy} \Leftrightarrow A \in A_2|_{Uses}$*. Two artifacts A and* A_2 *are equivalent iff their identifiers are equivalent, i.e.* $A \equiv A_2 \Leftrightarrow A|_{Identifier} \equiv A_2|_{Identifier}$*. We use* ε *to denote the empty artifact* $\varepsilon = (-, \emptyset, \emptyset)$*. We denote an artifact by its identifier preceded by* `#`*, i.e. the artifact representing class* `Canvas` *is denoted as* `#Canvas`*.*

An artifact can reference other artifacts that it depends on, e.g. artifacts representing method calls or field accesses would depend on the method or the field respectively.

Definition 3 *A* node $N \in \mathbb{AN}$ *(given a universe of nodes* $\mathbb{AN}$*) is a six-tuple* $(\mathtt{SequenceNumber}, \mathtt{Artifact}, \mathtt{Type}, \mathtt{Children}, \mathtt{Ordered}, \mathtt{SequenceGraph})$ *where* $\mathtt{SequenceNumber} \in \mathbb{N}$*,* $\mathtt{Artifact} \in \mathbb{A}$*,* $\mathtt{Type} \in \{0,1\}$*.* `Children` *is the set of child nodes.* $\mathtt{Ordered} \in \{0,1\}$ *and* `SequenceGraph` *is a sequence graph. Two nodes* N_1 *and* N_2 *are equivalent iff their sequence numbers are equal and their artifacts are equivalent, i.e.* $N_1 \equiv N_2 \Leftrightarrow (N_1|_{SequenceNumber} = N_2|_{SequenceNumber} \wedge N_1|_{Artifact} \equiv N_2|_{Artifact})$*. We use* ϑ *to denote the empty node* $\vartheta = (\mathtt{0}, \varepsilon, \mathtt{0}, \emptyset, \mathtt{0}, -)$*. We denote nodes by the identifier of their artifact preceded by \$, i.e. the node containing the artifact representing class* `Canvas` *is denoted as* \$`Canvas`*. We call the trees made up of such nodes* artifact trees*.*

A node N can be of type *solid* (i.e. $N|_{Type} = 1$) or of type *weak* (i.e. $N|_{Type} = 0$). A node of type weak inside of an artifact tree is just a placeholder to keep a path to its children, e.g. to identify a statement we also need to keep track of the containing method and class. A leaf node always has to be a solid node, i.e. a weak node only occurs if at least one of its subtrees contains a solid node. We do not consider artifacts that are contained in weak nodes as part of the tree. This will be used later during the traceability extraction to express for example that a statement but not its containing method are part of a trace.

Additionally a node N can either be an *ordered node* (i.e. $N|_{Ordered} = 1$) or an *unordered node* (i.e. $N|_{Ordered} = 0$). For ordered nodes the order of their children matters which is for example the case for nodes representing methods as the order of statements matters. The children of an ordered node hence have a specific order that is maintained in the node's *sequence graph* ($N|_{SequenceGraph}$) which will be explained later.

An artifact does not have to be unique to a product and also not within one product. An artifact can occur many times at many different places inside of many tress. For example an artifact representing a statement like `lines.clear();` could occur in many methods or even multiple times inside of one method. So when asking for a specific instance of an artifact and whether it is part of a product or not (e.g. should statement `lines.clear();` go into line 1 of method `wipe();` or not?) then also its position within the artifact tree has to be considered. So we are not asking if an artifact `#lines.clear();` should be included but rather if the concrete instance contained in the node \$`lines.clear`(); should be included.

Based on these we define a *product* as follows:

Definition 4 *A* product *$P \in \mathbb{P}$ is a two-tuple* ($\texttt{Features}, \texttt{RootNode}$) *where* `Features` *is the set of features that a product provides and* $\texttt{RootNode} = (\texttt{0}, \varepsilon, \texttt{1}, \texttt{Nodes}, \texttt{0}, -)$ *with* `Nodes` *being the set of nodes that contain the highest level artifacts that implement the product. We use $\mathbb{F}$ to denote the universe of features and $\mathbb{P}$ to denote the universe of products.*

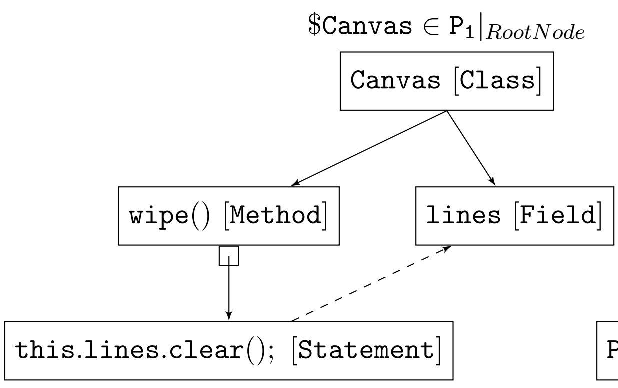
An example for a product would be $\texttt{P}_1 = (\{\texttt{BASE}, \texttt{LINE}, \texttt{WIPE}\}, RootNode)$ with $RootNode|_{Children} = \{\texttt{\$Canvas}, \texttt{\$Line}, \texttt{\$Main}\}$.

Examples of the artifact trees for $\texttt{P}_1$ are shown in Figure 3.2. The tree representing the class `Canvas` in product $\texttt{P}_1$ is shown in Figure 3.2a. As an identifier for classes and fields simply their respective names are used, methods are identified by their Java signature and statements simply by their Java source representation. Node $`Canvas` representing the class `Canvas` is an unordered node with node $`wipe`() representing a method and $`lines` representing a field as children. Node $`wipe`() is an ordered node. Its children represent statements (e.g. $`lines.clear`()) and their order matters. The artifact in node $`lines.clear`() references the artifact in node $`lines` as it represents a statement accessing that field. This is denoted by a dashed arrow. The tree for class `Main` in product $\texttt{P}_1$ is shown in Figure 3.2c. It contains a field `toolPanel` and a method `initContentPane`, which again is an ordered node that contains two statements `toolPanel.add(lineButton);` and `toolPanel.add(wipeButton)`. And lastly Figure 3.2b shows the tree representing the code for class `Line`. It has as a child only its constructor which is just a method identified by its signature, meaning only the types and order of its parameters are taken into account and not their names. The actual parameters, including the names, are then just treated like statements that precede all the actual statements.

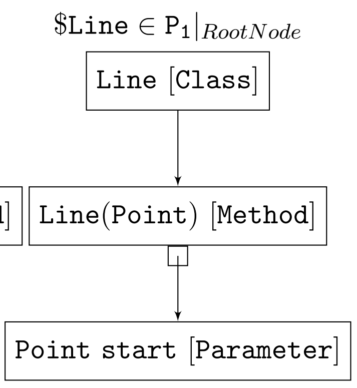
The source code for the remaining two products $\texttt{P}_2$ and $\texttt{P}_3$ is shown in Figures 3.3 and 3.4.

Note that when we use the operator $=$ for *artifacts* and *sequence graphs* we assume *reference* semantics, i.e. $=$ sets or compares the reference and does not copy or compare the contents. This is the same semantic as in Java when assigning or comparing objects.

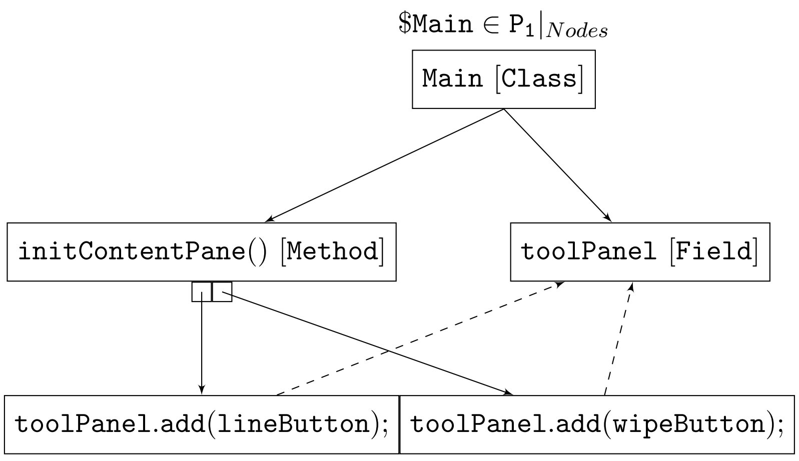
We define the following operations that exist for sets also for artifact trees:



(a) Class `Canvas`



(b) Class `Line`


$Main ∈ P₁|Nodes
Main [Class]
initContentPane() [Method]
toolPanel [Field]
toolPanel.add(lineButton);
toolPanel.add(wipeButton);

(c) Class `Main`

Figure 3.2: Artifact Trees for Product $\texttt{P}_1$

Definition 5 *To express that a node N is contained in an artifact tree with root node Root we write $N \in Root$.*

$N \in Root \Leftrightarrow N = Root \vee (\exists n \in Root|_{Children} : N \in n)$.

Definition 6 *To express that a node N_1 is a subset of another Node N_2 we write $N_1 \subseteq N_2$.*

$N_1 \subseteq N_2 \Leftrightarrow N_1 \equiv N_2 \wedge (N_1|_{Type} \Rightarrow N_2|_{Type}) \wedge \forall n_1 \in N_1|_{Children} : \exists n_2 \in N_2|_{Children} : n_1 \equiv n_2 \wedge n_1 \subseteq n_2$.

Definition 7 *The* cardinality *$|N|$ of a node N is the number of solid nodes contained in the artifact tree of which N is the root.*

$|N| = N|_{Type} + \sum_{n \in N|_{Children}} |n|$.

```
class Canvas {
  List<Line> lines;
  void setColor(String c){...}
  ...
}
class Line {
  Line(Color c, Point start){...}
  ...
}
class Main extends JFrame{
  initContentPane() {
    toolPanel.add(lineButton);
    toolPanel.add(colorsPanel);
  } ...
}
```

Figure 3.3: Source Code Snippets for Product $\mathtt{P}_2$ (`BASE`, `LINE`, `COLOR`)

For example the cardinality $|N_1|$ of node N_1 in Figure 3.5a is 4 since all 4 nodes are solid. We depict solid nodes with a solid box and weak nodes with a dotted box.

Definition 8 *The* full cardinality $\|N\|$ *is the number of nodes contained in the artifact tree of which N is the root, regardless whether these nodes are weak or solid.*

$$\|N\| = 1 + \sum_{n \in N|_{Children}} \|n\|.$$

The full cardinality $\|N_3\|$ of node N_3 in Figure 3.5c is 3 because the artifact tree of which N_3 is the root consists of 1 weak and 2 solid nodes.

Definition 9 *The intersection $N = N_1 \cap N_2$ of two nodes N_1 and N_2 has similar semantics as for sets and is defined as follows:*

The nodes must be equivalent, sequence graphs must match and they must either all be ordered or all unordered: $N_1 \equiv N_2 \wedge N|_{Ordered} = N_1|_{Ordered} = N_2|_{Ordered} \wedge N|_{SequenceGraph} = N_1|_{SequenceGraph} = N_2|_{SequenceGraph}$

N references the same artifact as N_1: $N|_{Artifact} = N_1|_{Artifact}$.

For N to be solid both N_1 and N_2 must be solid: $N|_{Type} = N_1|_{Type} \wedge N_2|_{Type}$.

Children of N must have an equivalent node among both N_1's children and N_2's children and must either be solid themselves or contain at least

```
class Canvas {
  List<Line> lines;
  List <Rectangle> rects;
  void setColor(String c){...}
  ...
}
class Line {
  Line(Color c, Point start){...}
  ...
}
class Rect {
  Rect(Color c, int x, int y){...}
  ...
}
class Main extends JFrame{
  initContentPane() {
    toolPanel.add(lineButton);
    toolPanel.add(rectButton);
    toolPanel.add(colorsPanel);
  } ...
}
```

Figure 3.4: Source Code Snippets for Product $\texttt{P}_3$ (`BASE`, `LINE`, `RECT`, `COLOR`)

one solid descendant: $n \in N|_{Children} \Leftrightarrow \exists n_1 \in N_1|_{Children}, n_2 \in N_2|_{Children} : n_1 \equiv n_2 \wedge n = n_1 \cap n_2 \wedge (n|_{Type} = 1 \vee |n|_{Children}| > 0)$.

The intersection of the `$Canvas` node N_1 of product $\texttt{P}_1$ and the `$Canvas` node N_2 of product $\texttt{P}_2$ is shown in Figure 3.5e. The two trees have their root node `$Canvas` in common and the child node `$lines`, therefore the tree resulting from the intersection contains exactly these two nodes, and both are solid since they were both solid in each of the two intersected trees.

Definition 10 *The union* $N = N_1 \cup N_2$ *of two nodes* N_1 *and* N_2 *has similar semantics as for sets and is defined as follows:*

The nodes must be equivalent, sequence graphs must match and they must either all be ordered or all unordered: $N_1 \equiv N_2 \wedge N|_{Ordered} = N_1|_{Ordered} = N_2|_{Ordered} \wedge N|_{SequenceGraph} = N_1|_{SequenceGraph} = N_2|_{SequenceGraph}$

N *references the same artifact as* N_1*:* $N|_{Artifact} = N_1|_{Artifact}$.

For N *to be solid at least one of* N_1 *or* N_2 *must be solid:*
$N|_{Type} = N_1|_{Type} \vee N_2|_{Type}$.

For every child of N *one of the following conditions must hold:*
$n \in N|_{Children} \Leftrightarrow$

$$\exists n_1 \in N_1|_{Children} : (\nexists n_2 \in N_2|_{Children} : n_1 \equiv n_2) \wedge n = n_1$$
$$\vee\ \exists n_2 \in N_2|_{Children} : (\nexists n_1 \in N_1|_{Children} : n_1 \equiv n_2) \wedge n = n_2$$
$$\vee\ \exists n_1 \in N_1|_{Children}, n_2 \in N_2|_{Children} : (n_1 \equiv n_2 \wedge n = n_1 \cup n_2)$$

The union N_6 of the two trees $N_3 = N_1 \setminus N_2$ (Figure 3.5c) and $N_4 = N_2 \setminus N_1$ (Figure 3.5d) is shown in Figure 3.5f. N_6 contains all the nodes from N_3 and N_4. If a node is weak in both N_3 and N_4 it is also weak in N_6, if it appeared solid in any of the two in will be solid in N_6. Node `$Canvas` remains weak as it was weak in both N_3 and N_4. The descendants of `$Canvas` from N_3 and N_4 all appear solid in the union as they are all solid in any of N_3 or N_4.

Definition 11 *The minus operation $N = N_1 \setminus N_2$ of two nodes N_1 and N_2 has similar semantics as for sets and is defined as follows:*

The nodes must be equivalent, sequence graphs must match and they must either all be ordered or all unordered: $N_1 \equiv N_2 \wedge N|_{Ordered} = N_1|_{Ordered} = N_2|_{Ordered} \wedge N|_{SequenceGraph} = N_1|_{SequenceGraph} = N_2|_{SequenceGraph}$

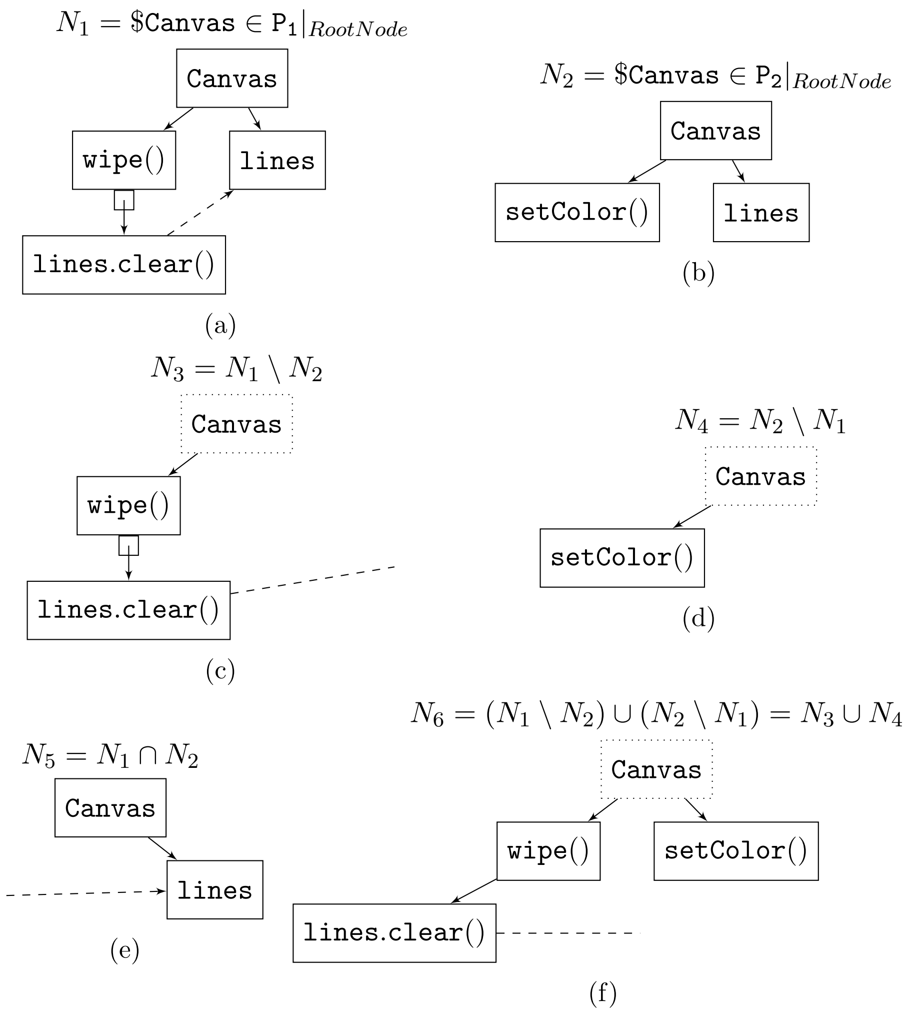
N references the same artifact as N_1: $N|_{Artifact} = N_1|_{Artifact}$.

For N to be solid N_1 must be solid and N_2 must be weak:
$N|_{Type} = N_1|_{Type} \wedge \neg N_2|_{Type}$.

For children of N the following must hold:
$n \in N|_{Children} \Leftrightarrow$

$$\exists n_1 \in N_1|_{Children} : (\nexists n_2 \in N_2|_{Children} : n_1 \equiv n_2) \wedge n = n_1$$
$$\vee\ \exists n_1 \in N_1|_{Children}, n_2 \in N_2|_{Children} : n_1 \equiv n_2 \wedge n = n_1 \setminus n_2 \wedge (n|_{Type} = 1 \vee |n|_{Children}| > 0)$$

The minus operation $N_3 = N_1 \setminus N_2$ is shown in Figure 3.5c. Any solid node in N_2 that also appears solid in N_1 appears weak in N_3 in case it has a solid decendant in N_3 or it does not appear at all. Node `$wipe`() appears solid in N_1 and not in N_2 which is why it is solid in N_3. `$Canvas` is solid in both N_1 and N_2 and since it has solid descendants (`$wipe`()) it appears weak in N_3. Node `$line` is solid in both and since it does not have any children does not appear in N_3 at all.



$N_1 = \texttt{\$Canvas} \in \texttt{P}_1|_{RootNode}$
Canvas
wipe()
lines
lines.clear()
(a)
$N_2 = \texttt{\$Canvas} \in \texttt{P}_2|_{RootNode}$
Canvas
setColor()
lines
(b)
$N_3 = N_1 \setminus N_2$
Canvas
wipe()
lines.clear()
(c)
$N_4 = N_2 \setminus N_1$
Canvas
setColor()
(d)
$N_5 = N_1 \cap N_2$
Canvas
lines
(e)
$N_6 = (N_1 \setminus N_2) \cup (N_2 \setminus N_1) = N_3 \cup N_4$
Canvas
wipe()
setColor()
lines.clear()
(f)

Figure 3.5: Minus (c)(d), Intersection (e) and Union (f) operations between `$Canvas` Nodes of Product $\texttt{P}_1$ (a) and Product $\texttt{P}_2$ (b)

Chapter 4

Extraction

This chapter describes the extraction of information out of sets of related product variants. Mainly two types of information are extracted:

- Traceability: mappings between modules and their implementing artifacts.
- Dependencies: dependencies between traces which represent a simple form of variability model.

These two steps are described in the following sections. The extraction process is incremental, meaning it can start out with just one single product variant and be refined any time with additional product variants as soon as they become available.

Usually traceability information is extracted as a mapping between single features and their implementing artifacts. However, it does not suffice to just consider single features when deciding what artifacts are required to implement a product, because also the interactions of features require artifacts to implement them. Meaning, whether certain artifacts are required for implementing a product variant can also depend on certain combinations of features being present. Furthermore, when the implementing artifacts are source code then the granularity at which traceability is extracted is commonly on the level of classes or methods, neither is sufficient for our purpose though. This level of granularity is too coarse. A lot of variability occurs at the level of statements. And with that comes the additional challenge that for statements the question is not only whether they are required for the implementation of a feature (or a feature interaction) but also in what order they should be included. A certain combination of features could for example be reflected in the implementation by simply changing the order of already existing artifacts. So the traceability information we aim to extract does no longer just include the presence or absence of artifacts but also their ordering.

The traceability extraction therefore has two problems to solve:

- Traceability: determining whether an artifact traces to a module or not.
- Ordering: determining what the order relations between children of ordered nodes are.

4.1 Traceability

To express whether artifacts implement a single feature or an interaction between features we introduce modules that are inspired by a variation of *Feature Algebra* from Liu et al. [Liu et al., 2006]. A detailed description is presented in [Linsbauer et al., 2013]

There are two kinds of modules: `base` modules and `derivative` modules. Modules are simply sets of features.

Definition 12 *A* base module *represents artifacts that are responsible for the implementation of a single feature and that are present in every product that implements this feature, independently of any other features. We denote a base module with the feature's name in lowercase. A base module* `b` *of a feature* `B` *is represented as the set* $\mathtt{b} = \{\mathtt{B}\}$ *only containing the feature* `B`*.*

The base module for feature `LINE` would for example be `line` and it would, among others, map to the field `lines` of class `Canvas` (shown in Figure 3.1 in Line 2) corresponding to Node `$lines` in Figure 3.2a. This field is present in every product that has feature `LINE` selected.

Definition 13 *A* derivative module *or simply* derivative*, denoted as* $\delta^{\mathtt{n}}(\mathtt{F_0}, \mathtt{F_1}, ..., \mathtt{F_n})$*, represents artifacts that implement feature interactions, where* F_i *is* $\mathtt{F} \in \mathbb{F}$ *if feature* `F` *is selected or* $\neg\mathtt{F}$ *if* `F` *is not selected, and* n *is the order of the derivative. A derivative module of order* n *represents the interaction of* $n + 1$ *features. A derivative module* $d = \delta^{\mathtt{n}}(\mathtt{F_0}, \mathtt{F_1}, ..., \mathtt{F_n})$

is represented as the set containing all the interacting features, i.e. $d = \{F_0, F_1, ..., F_n\}$.

The derivative module representing the interaction of features `LINE` and `WIPE` would for example be $\delta^1(\texttt{line}, \texttt{wipe})$ of order 1. It would map to the statement `this.lines.clear();` as shown in Figure 3.2a in Line 4 corresponding to Node $`this.lines.clear`(); in Figure 3.2a.

Modules can only add artifacts. To model the removal of artifacts we allow negative features in derivatives. A negative feature ¬`F` is used to express that feature `F` must not be selected in a product to include certain artifacts. An example would be $\delta^1(\texttt{line}, \neg\texttt{color})$ which represents the artifacts that should occur in a product if feature `LINE` is selected while feature `COLOR` is deselected (i.e. negative). It represents the feature interaction between `LINE` and ¬`COLOR`. The corresponding artifacts would be the constructor `Line(Point start) {...}` in Line 9 of Figure 3.1 represented by node $`Line`(`Point`) in Figure 3.2b.

We use $\mathbb{M}$ to denote the universe of modules.

To describe the mappings between modules and artifacts we introduce *associations.*

Definition 14 *Given a universe of associations* $\mathbb{AS}$ *an association* $AS \in \mathbb{AS}$ *is a 5-tuple* $(\texttt{Modules}, \texttt{Max}, \texttt{All}, \texttt{Not}, \texttt{RootNode})$ *where* $\texttt{Modules} \in \mathbb{M}, \texttt{Max} \in \mathbb{M}, \texttt{All} \in \mathbb{M}$ *and* $\texttt{Not} \in \mathbb{M}$ *are sets of modules and* $\texttt{RootNode} = (-\texttt{1}, \varepsilon, \vartheta, \texttt{Children}, \texttt{0}, -)$ *is the root of the tree containing the relevant artifacts. An association maps modules to artifacts. Alternatively we say that these modules trace to these artifacts.*

We say that an artifact is contained in an association if its containing node is part of the association's artifact tree.

In [Linsbauer et al., 2013] we found that not all artifacts have a unique trace, i.e. they can trace to multiple disjunctive modules. For example an artifact that traces to module `line` *or* module `rect` is included in any product that contains either of these modules. This means that products can have artifacts in common even though they do not have any features in common. For this purpose, in addition to the ordinary set of modules `Modules` as it was already presented in [Linsbauer et al., 2013], there are now three additional sets of modules in an association: `Max`, `All` and `Not`.

Assume a given set of products $P \subseteq \mathbb{P}$ and an already extracted association $AS \in \mathbb{AS}$.

- `Modules`: the set of modules that all products that contain the corresponding artifacts have in common. This is based on the assumption that artifacts that products have in common trace to the modules these products have in common. However, this is only true for artifacts that have a unique trace, i.e. trace to only one module. For artifacts with a non-unique trace this set will be the empty set. The artifacts in AS could trace to exactly one module $m \in AS|_{Modules}$.

  $AS|_{Modules} = \{m \mid \forall p \in P : AS|_{RootNode} \subseteq P|_{RootNode} \Rightarrow m \in FeaturesToModules(P|_{Features})\}$.

- `All`: the set containing all modules of all products that contain the corresponding artifacts. This set contains all modules that the corresponding artifacts have ever been associated with in any of the products.

  $AS|_{All} = \{m \mid \exists p \in P : m \in featuresToModules(P|_{Features}) \wedge AS|_{RootNode} \subseteq P|_{RootNode}\}$.

- `Not`: the set containing all modules of all products that do not contain the corresponding artifacts. The artifacts in AS do not trace to any module $m \in AS|_{Not}$.

  $AS|_{Not} = \{m \mid \exists p \in P : m \in featuresToModules(P|_{Features}) \wedge AS|_{RootNode} \nsubseteq P|_{RootNode}\}$.

- `Max`: this set simply contains all modules in `All` that are not also in `Not`, i.e. the modules that the corresponding artifacts can at most trace to. The artifacts in AS could potentially trace to any set of modules $M \subseteq AS|_{Max}$.

  $AS|_{Max} = AS|_{All} \setminus AS|_{Not}$.

To work with modules we need to be able to negate sets of features, compute a set of modules out of a set of features and update module sets with new features. This is done using the functions shown below. Assume a set of selected (i.e. positive) features $F \subseteq \mathbb{F}$ and a set F_{all} of all currently known features in a domain with $F \subseteq F_{all}$.

Definition 15 *$negateFeatures : 2^{\mathbb{F}} \mapsto 2^{\{\neg f | f \in \mathbb{F}\}}$ is a function that computes for a set of features the set with the same features negated.*

$$\bar{F} = negateFeatures(F) = \{\neg f \mid f \in F\}$$

Definition 16 *$featuresToModules : 2^{\mathbb{F}} \times 2^{negateFeatures(\mathbb{F})} \mapsto 2^{\mathbb{M}}$ computes from a given set of selected (i.e. positive) features and a set of unselected (i.e. negative) features the corresponding set of modules that can be formed.*

$$M = featuresToModules(F, \bar{F}) = \{p \cup n \mid p \in 2^F \setminus \emptyset \wedge n \in 2^{\bar{F}}\}$$

with $\bar{F} = negateFeatures(F_{all} \setminus F)$.

Definition 17 *$updateModules : 2^{\mathbb{M}} \times 2^{negateFeatures(\mathbb{F})} \mapsto 2^{\mathbb{M}}$ updates a set of modules with a set of negative features that were previously unknown in the domain and hence not contained in the modules.*

$$M' = updateModules(M, \bar{F}) = \{m \cup n \mid m \in M \wedge n \in 2^{\bar{F}}\}$$

with $\bar{F} = negateFeatures(F_{all} \setminus F)$.

The algorithm for extracting associations (i.e. traceability) from a product P, a set of all currently known features in a domain F and an already existing set of associations AS from already processed products is shown in Algorithm 4.1. It is based on [Linsbauer et al., 2013]. Details and optimizations like not adding empty associations or merging associations that do not contain any artifacts are omitted for simplicity.

For every input product $p \in P$ first its corresponding initial association is computed in Line 9. In case the product introduced new features to the domain the existing associations $a \in AS$ as well as the set of features F are updated in Lines 11-20.

In Line 24 all the ordered nodes in the new association a_{new} are aligned (i.e. their sequence numbers are updated) to the respective nodes contained in the already existing associations and their sequence graphs are updated to incorporate new information about ordering. The function $alignAndSequenceTree : \mathbb{AN} \times \mathbb{AN} \mapsto \mathbb{AN}$ is responsible for taking care of the ordering of children of ordered nodes. It will be explained in the following section.

Algorithm 4.1 Traceability Extraction Algorithm

```
1  Input: A set of Products (P),
2         A set of features (F),
3         A set of associations (AS)
4  Output: A List of refined associations assocs,
5          The updated set of features.
6
7  for $p$ in $P$ begin
8   {Compute initial association for Product $p$}
9   $a_{new} := (featuresToModules(p|_{Features}, F \setminus p|_{Features}), p|_{RootNode})$
10
11  {Update Associations in case of new features}
12  if $p|_{Features} \not\subseteq F$ then
13   for $a$ in AS begin
14    $AS|_{Modules} := updateModules(AS, p|_{Features} \setminus F)$
15    $AS|_{All} := updateModules(AS, negateFeatures(p|_{Features} \setminus F)$
16    $AS|_{Not} := updateModules(AS, p|_{Features} \setminus F)$
17    $AS|_{Max} := updateModules(AS, p|_{Features} \setminus F)$
18   endfor
19   $F := F \cup p|_{Features}$
20  endif
21
22  {Align and Sequence $a_{new}$ to the associations AS}
23  for $a$ in AS begin
24   $alignAndSequenceTree(a|_{RootNode}, a_{new}|_{RootNode})$
25  endfor
26
27  {Refine AS using the association $a_{new}$}
28  $assocs := \emptyset$
29  for $a$ in AS begin
30   $(a_l, a_{int}, a_r) := compare(a, a_{new})$
31   $a_{new} := a_r$
32   $assocs := assocs \cup \{a_l, a_{int}\}$
33  endfor
34  $assocs := assocs \cup \{a_{new}\}$
35  AS $= assocs$
36 endfor
37 return AS, F
```

The algorithm then performs comparisons between associations' module sets and artifact trees in Lines 29-33. For that it uses the function *compare* : $\mathbb{AS} \times \mathbb{AS} \mapsto \mathbb{AS} \times \mathbb{AS} \times \mathbb{AS}$ which gets as input an already processed association A and a new association A_{new}. It performs comparisons for the two associations' module sets and artifact trees as follows:

Definition 18 $(A_l, A_{int}, A_r) = compare(A, A_{new}) \Leftrightarrow$
The artifact trees are compared in terms of common and unique artifacts:

- $A_{int}|_{RootNode} = A|_{RootNode} \cap A_{new}|_{RootNode}$
- $A_l|_{RootNode} = A|_{RootNode} \setminus A_{int}|_{RootNode}$
- $A_r|_{RootNode} = A_{new}|_{RootNode} \setminus A_{int}|_{RootNode}$

Modules are compared in the exact same way. Common artifacts trace to common modules. Artifacts that are unique to an association trace to modules that are unique to that association.

- $A_{int}|_{Modules} = A|_{Modules} \cap A_{new}|_{Modules}$
- $A_l|_{Modules} = A|_{Modules} \setminus A_{int}|_{Modules}$
- $A_r|_{Modules} = A_{new}|_{Modules} \setminus A_{int}|_{Modules}$

Common artifacts were in contact with all modules that A *and* A_{new} *were ever in contact with. Nothing new about modules these artifacts do not trace to can be derived.*

- $A_{int}|_{All} = A|_{All} \cup A_{new}|_{All}$
- $A_{int}|_{Not} = A|_{Not}$

Artifacts that are not common do not trace to any of the modules the respective other association has ever been in contact with:

- $A_l|_{Not} = A|_{Not} \cup A_{new}|_{All}$
- $A_r|_{Not} = A_{new}|_{Not} \cup A|_{All}$

The maximal set of modules that artifacts can trace to are computed:

- $A_l|_{Max} = A_l|_{All} \setminus A_l|_{Not}$
- $A_{int}|_{Max} = A_{int}|_{All} \setminus A_{int}|_{Not}$
- $A_r|_{Max} = A_r|_{All} \setminus A_r|_{Not}$

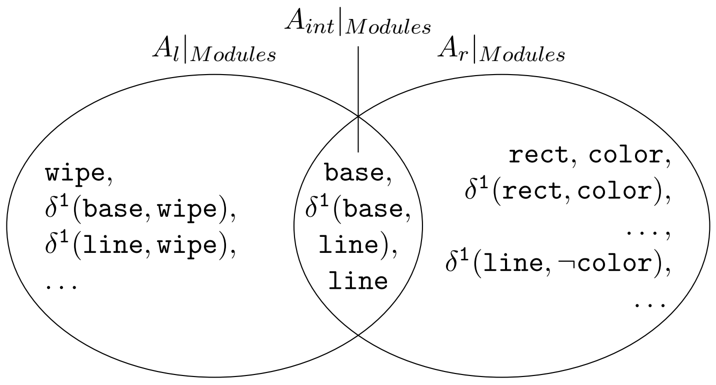

$A_{int}|_{Modules}$
$A_l|_{Modules}$
$A_r|_{Modules}$
wipe, δ^1(base, wipe), δ^1(line, wipe), ...
base, δ^1(base, line), line
rect, color, δ^1(rect, color), ..., δ^1(line, ¬color), ...

Figure 4.1: Comparison of Modules of Association A_1 and Association A_3

The necessary operations (i.e. intersection and minus operation) have been defined for artifact trees in Chapter 3 along with examples. For modules this are simple set operations. The comparison of $A_1|_{Modules}$ with A_1 being the initial association for product $\mathtt{P}_1$ and $A_3|_{Modules}$ with A_3 being the initial association for $\mathtt{P}_3$ is shown in Figure 4.1.

After the trace extraction every node appears solid exactly in one of the associations and may appear weak in any of the other associations as parents for solid descendants. This means that every node (and therefore its contained artifact) are part of exactly one association. Note that an association can represent traces to multiple modules in case of a non-unique trace.

4.2 Ordering

Before ordered nodes (i.e. nodes N with $N|_{Ordered} = 1$) can be compared their sequence graphs first have to be aligned to each other and then their sequence graphs have to be properly updated according to the computed alignment. In an ordered node every child node has a sequence number that is unique in the corresponding sequence graph. The corresponding node's sequence graph represents a partial order over these sequence numbers.

Definition 19 *A* sequence graph *SG is a 3-tuple $(Sequenced, IDs, Order)$. $Sequenced \in \{0, 1\}$ tells whether the sequence graph has already been aligned with another one. IDs is a function $\mathbb{N} \mapsto \mathbb{A}$ that maps sequence numbers to artifacts. Every sequence number in the sequence graph has exactly one entry* *in IDs. Order is a non-strict partial order over a set of sequence numbers $S = \{id \mid \exists(id, a) \in SG|_{IDs}\} \subseteq \mathbb{N}$ and therefore satisfies the following conditions:*

- *Reflexive:* $\forall a \in S : (a, a) \in SG$
- *Antisymmetric:* $\forall a, b \in S : (a, b) \in SG \wedge (b, a) \in SG \Rightarrow a = b$
- *Transitive:* $\forall a, b, c \in S : (a, b) \in SG \wedge (b, c) \in SG \Rightarrow (a, c) \in SG$

Definition 20 *An alignment maps sequence numbers of one ordered node's sequence graph $N_2|_{SequenceGraph}$ to sequence numbers of another node's sequence graph $N_1|_{SequenceGraph}$. An alignment $a = align(N_1, N_2)$ for two ordered nodes N_1 and N_2 must fulfill the following conditions:*

- *A sequence number in N_2 may map to another sequence number in N_1 only if they represent the same artifact.* $\forall(id_1, art_1) \in N_1|_{SequenceGraph}|_{IDs}, (id_2, art_2) \in N_2|_{SequenceGraph}|_{IDs} : a(id_2) = id_1 \Rightarrow art_2 \equiv art_1$
- *The two sequence graphs must not contradict each other after the alignment.* $\forall(x, y) \in N_2|_{SequenceGraph}|_{Order} : (a(x), a(y)) \notin N_1|_{SequenceGraph}|_{Order}$
- *For every sequence number there must be at most one mapping in a and no two sequence numbers may map to the same sequence number.* $\forall(x, y) \in a, (x', y') \in a : x = x' \Leftrightarrow y = y'$
- *For every sequence number in N_2's sequence graph there must be exactly one mapping in a.* $\forall(id, art) \in N_2|_{SequenceGraph}|_{IDs} : \exists(x, y) \in a : x = id$

There are many possible alignments that fulfill these conditions. Which one to choose is determined by a cost function that should be minimized. Among all the possible alignments one with the lowest cost is chosen. We use the following cost function:

$$cost(a, N_1) = |\{x \mid \exists(x, y) \in a : \nexists(id, art) \in N_1|_{SequenceGraph}|_{IDs} : y = id\}|$$

This function minimizes the number of new sequence numbers used, i.e. it maximizes the number of reused sequence numbers from N_1. Any other cost function can be chosen that for examples considers semantic knowledge about the kind of artifacts, e.g. in case of artifacts that represent Java source code the cost function could apply a penalty to alignments according to which variables would be used before they are initialized.

The function $a = align_{minCost}(N_1, N_2) \Leftrightarrow a = align(N_1, N_2) \wedge \nexists a' = align(N_1, N_2) : cost(a', N_1) < cost(a, N_1)$ provides such an alignment.

Given such an alignment $a = align_{minCost}(N_1, N_2)$ the sequence graphs of N_1 and N_2 and the sequence numbers of their child nodes have to be updated so that they match and the nodes can be compared using the operations defined in the previous Chapter.

Definition 21 *To compute the updated sequence graph SG' the function sequence is used. It merges the IDs and the Order of two sequence graphs with respect to the alignment a and sets the sequence graph to $Sequenced = 1$.*

$SG' = sequence(a, SG_1, SG_2) \Leftrightarrow$

- *The sequence graph has been sequenced: $SG'|_{Sequenced} = 1$.*
- *The IDs from SG_1 are used directly and the IDs from SG_2 are added according to the alignment a: $SG'|_{IDs} = \{(id, art) \mid (id, art) \in SG_1|_{IDs} \vee (\exists (id', art) \in SG_2|_{IDs} : a(id') = id)\}$*
- *The Order is merged just as the IDs are but in addition it also has to remain transitive: $\{(x, y) \mid (x, y) \in SG_1|_{Order} \vee (a(x), a(y)) \in SG_2|_{Order}\} \subseteq SG'|_{Order}$ and $\forall x, y, z : (x, y) \in SG'|_{Order} \wedge (y, z) \in SG'|_{Order} \Rightarrow (x, z) \in SG'|_{Order}$.*

Example: Consider an ordered node N_2 that should be aligned to another ordered node N_1. Both nodes contain an artifact `#m` representing the same method `m` but from different product variants, which is why their children (i.e. the statements of method `m`) are different. The statements for N_1 and N_2 are shown in Figure 4.2a and Figure 4.2b respectively.

N_1's sequence graph looks as follows:

$N_1|_{SequenceGraph}|_{Sequenced} = 0$

$N_1|_{SequenceGraph}|_{IDs} = \{(1, \texttt{\#i++}), (2, \texttt{\#j++}), (3, \texttt{\#k=i+j})\}$ and

$N_1|_{SequenceGraph}|_{Order} = \{(1, 2), (2, 3), (1, 3)\}$.

N_2's sequence graph looks like this:

$N_2|_{SequenceGraph}|_{Sequenced} = 0$

$N_2|_{SequenceGraph}|_{IDs} = \{(1, \texttt{\#i++}), (2, \texttt{\#j--}), (3, \texttt{\#k=i+j})\}$ and

$N_2|_{SequenceGraph}|_{Order} = \{(1, 2), (2, 3), (1, 3)\}$.

The alignment $a = align_{minCost}(N_1, N_2) = \{(1, 1), (3, 3), (2, 4)\}$ cannot map 2 to any sequence number of N_1 because $\nexists(id, art)\ inN_1|_{SequenceGraph}|_{IDs}$: $art = N_2|_{SequenceGraph}|_{IDs}(2)$ and therefore assigns the new sequence number 4 to it.

The updated sequence graph $SG = sequence(a, N_1, N_2)$ then looks as follows:

- It is now sequenced: $SG|_{Sequenced} = 1$.
- It contains an additional sequence number: $SG|_{IDs} = \{(1, \texttt{\#i++}), (2, \texttt{\#j++}), (3, \texttt{\#k=i+j}), (4, \texttt{\#j--}\}$.
- The partial order is updated: $SG|_{Order} = \{(1, 2), (2, 3), (1, 3), (1, 4), (4, 3)\}$.

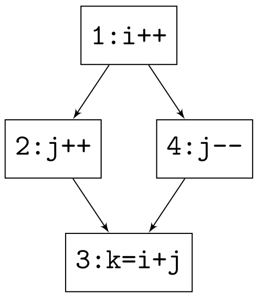
The new sequence graph SG is shown in Figure 4.2c. The order of statements `j++` and `j--` cannot be determined at this point because they appear in the same place but haven't appeared together yet.

```
1 i++;
2 j++;
3 k=i+j;
```

(a)

```
1 i++;
2 j--;
3 k=i+j;
```

(b)



(c)

Figure 4.2: Example Sequence Graph

What the function *alignAndSequenceTree* then does is it traverses the artifact trees and updates them by aligning all the matching ordered nodes to each other. The pseudo code for it is shown in Algorithm 4.2.

Algorithm 4.2 alignAndSequenceTree

```
Input: Node $N_1$
       Node $N_2$

{If nodes are ordered align them}
if $N_1|_{Ordered} = N_2|_{Ordered} = 1$ then
 if $N_2|_{sequenced} = 1$ then
  $N_1|_{SequenceGraph} := N_2|_{SequenceGraph}$
 else
  $a := align_{minCost}(N_1, N_2)$
  $SG := sequence(a, N_1, N_2)$
  $N_1|_{SequenceGraph} := SG$
  $N_2|_{SequenceGraph} := SG$
  {Update sequence numbers of child nodes of $N_2$}
  for $n in N_2|_{Children}$ begin
   $n|_{SequenceNumber} = a(n|_{SequenceNumber})$
  endfor
 endif
endif

{Align and sequence matching child nodes}
for $n_1 \in N_1|_{Children}$ begin
 for $n_2 \in N_2|_{Children}$ begin
  if $n_1 \equiv n_2$ then
   $alignAndSequenceTree(n_1, n_2)$
  endif
 endfor
endfor
```

4.3 Dependencies

There are two kinds of dependencies:

- *Structural* dependencies: a child node requires its parent node to be present. Without the parent (e.g. the encapsulating class) the child node (e.g. a method) can not be included.

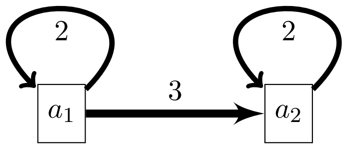
- *Cross-Tree* dependencies: an artifact may require another artifact that is located somewhere else in the tree to be present, e.g. a method call requires the called method to be present.

We weigh a structural dependency with weight 2 and a cross-tree dependency with weight 1. Based on this we compute dependencies between associations. A dependency matrix is a quadratic matrix with the rows and columns representing associations. In every cell $c_{i,j}$ the weighted sum of all dependencies of artifacts within association a_i to artifacts within association a_j is stored. Cells $c_{i,i}$ represent dependencies between artifacts within an association. Remember: an artifact is considered to be in an association (i.e. part of the trace) if its containing node is solid in the corresponding association's artifact tree.

As a simple example assume the artifact tree in Figure 3.5c as part of an association a_1 and the artifact tree in Figure 3.5e as part of another association a_2. The weight in cell $c_{1,2}$ would be 3 because the parent of `$wipe`() in a_1 is `$Canvas` in a_2 which adds a weight of 2, and `$lines.clear`() in a_1 accesses `$line` which also traces to a_2 and this adds a weight of 1 to a total of 3. The weight in cell $c_{2,1}$ on the other hand would be 0 because no artifact that is in a_2 depends on any artifact in a_1. The complete dependency matrix is shown in Figure 4.3a.

	a_1	a_2
a_1	2	3
a_2	0	2

(a)



(b)

Figure 4.3: Example Dependency Matrix (a) and corresponding Dependency Graph (b)

Such a dependency matrix can also be displayed as a dependency graph as shown in Figure 4.3b. Each node represents an association. The edges are labeled with their weight which is also reflected in their thickness.

4.4 Identifiers

A node $N \in \mathbb{AN}$ is identified by its sequence number $N|_{SequenceNumber} \in \mathbb{N}$ and its artifact $N|_{Artifact}$. An artifact $A \in \mathbb{A}$ is identified by its identifier $A|_{Identifier}$. Two artifacts with the same identifier are considered to be equivalent. Based on this the comparisons of artifact trees are performed. Currently identifiers are simply based on the names of the objects that the artifacts represent. E.g. an artifact representing a `class` in Java simply uses its name as an identifier as was done with the artifact `#Canvas` representing class `Canvas` in our draw example. This is very simple to implement and sufficient if the product variants used as input have not diverged from each other (i.e. not undergone independent evolutionary changes like bug fixes or feature changes etc.). However, if product variants have diverged strongly the extraction will often not be able to match artifacts that in fact should be matched. Consider for example one draw variant in which the class `Canvas` was renamed to `Base` to reflect the feature's name `BASE` that it corresponds to. Even though these two classes will be almost identical the extraction will not match them because they have different names.

This section will propose a way to mitigate this problem. It is part of our future work to implement and evaluate it.

Instead of using identifiers for artifacts based on the names of the objects these artifacts represent we want to use identifiers based on the actual similarity of these objects across product variants, e.g. similarity above a threshold of 90% according to an arbitrary metric (e.g. using some code-clone detection technique in the case of source code) should be considered a match. Consider again class `Canvas` in one draw variant (e.g. $\mathtt{P}_1$) and class `Base` in another (e.g. a new product $\mathtt{P}_6$) and assume they are 95% equivalent. These two artifacts would then receive the same identifier (e.g. simply the concatenation of their names: `CanvasBase`).

However, in order to be able to fully reconstruct all the products again artifact's names now have to be stored separately and become a part of the extraction process, since they now can also trace to modules. This can for example be achieved by simply treating the object's name as an artifact as well and putting it in an additional child node of the original artifact's node. Figure 4.4 shows the comparison of the artifact trees of class `Canvas` and class `Base` assuming only the class' name has changed and they are considered equivalent.

Of course this same principle can be applied on any level of granularity. Once two artifact nodes have been matched this same technique could also be applied to its children. It might for example make sense, once two classes have been matched, to also apply this technique on their methods to find out what methods should be matched. Because two methods that are almost identical would not be matched if only their signature has changed.

Of course the used similarity metric depends on the kind of objects the artifacts represent. It will be different for source code than it will be for models or any other artifacts. So every concrete artifact type will have to

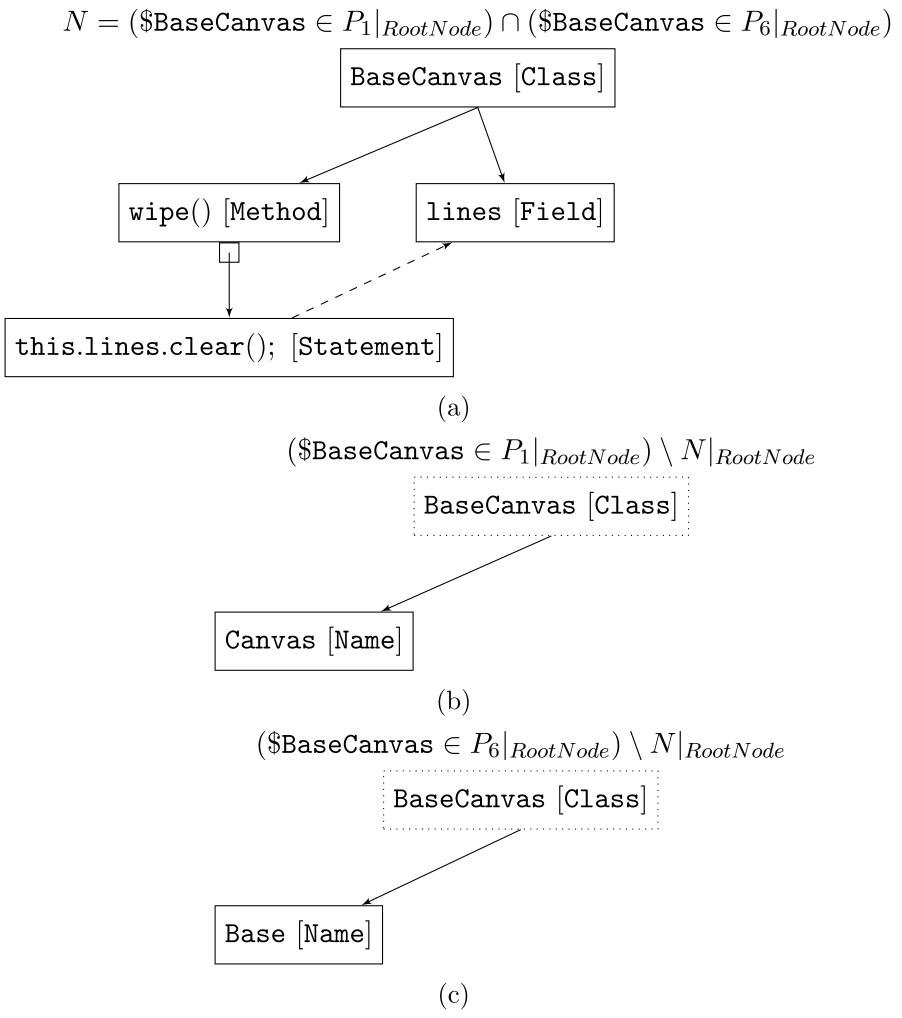

$N = (\texttt{\$BaseCanvas} \in P_1|_{RootNode}) \cap (\texttt{\$BaseCanvas} \in P_6|_{RootNode})$
BaseCanvas [Class]
wipe() [Method]
lines [Field]
this.lines.clear(); [Statement]
(a)
$(\texttt{\$BaseCanvas} \in P_1|_{RootNode}) \setminus N|_{RootNode}$
BaseCanvas [Class]
Canvas [Name]
(b)
$(\texttt{\$BaseCanvas} \in P_6|_{RootNode}) \setminus N|_{RootNode}$
BaseCanvas [Class]
Base [Name]
(c)

Figure 4.4: Artifact Tree for Matched Classes `Canvas` and `Base` with Identifier `BaseCanvas`

provide its own *comparison* metric.

The difficulty in this will be the incremental usage as later during the extraction process classes can be split up over several associations and measuring their similarity may require their complete reconstruction which could cost a lot of runtime. Additionally similarity not only between two but between an arbitrary number of instances of a class has to be considered. For example the following case is interesting: two classes `A` and `B` (each from another variant) are 90% equivalent and a third class `C` (again from another variant) is 90% equivalent to `A` but only 80% to `B`. Do they now receive the same identifier or not? In any case the order in which products are added to the database must not make a difference, meaning that the result in the end has to be the same for every order in which products are added to the database as long as the products are the same.

Chapter 5

Composition

The composition uses the previously extracted information to generate products. Based on a set of features that a product should have it selects associations and combines their artifacts into a product. This chapter gives a brief overview about the composition process. More details can be found in [Fischer, 2013].

The quality of the product of course depends on the quality of the previously extracted information. When composing a product that was among the input products for the extraction (i.e. a product with the same set of features) the information should suffice to perfectly reconstruct it. If, however, the newly composed product was not among the input products it may be incomplete. The composition automatically creates an as good as possible skeleton for the new product given the information it has available. This skeleton in combination with warnings provided by the composition can then be used by software engineers to manually complete the new product.

The composition maps a set of positive (i.e. selected) features F, a set all currently known features F_{all} and a set of associations AS to a product: $compose : 2^{\mathbb{F}} \times 2^{\mathbb{F}} \times 2^{\mathbb{AS}} \mapsto \mathbb{P}$.

The initial version of the composed product is computed as follows:
$P = compose(F, F_{all}, AS) = (F, RootNode)$
where $RootNode = \bigcup\limits_{assoc \in assocs} assoc|_{RootNode}$
with $assocs = \{assoc \mid m = featuresToModules(F, negateFeatures(F_{all} \setminus F)) \wedge (m \cap assoc|_{Modules} \neq \emptyset \vee assoc|_{Modules} = \emptyset \wedge m \cap assoc|_{Max} \neq \emptyset)\}$.

This version can then be automatically refined using the extracted dependency information and its manual completion can be guided using warnings as described in the following sections.

5.1 Warnings

A product can either be missing artifacts for modules that do not exist in the extracted associations because they were not part of any of the input products or it can contain surplus artifacts if modules could not be separated in the extracted associations because they never appeared separately in any of the input products. Additionally the order for artifacts that never appeared together might be uncertain. For all these cases the composition provides the appropriate warnings to guide the user through the manual completion process of the provided skeleton product.

Assume a set of extracted associations A, a set of positive features F a composed product should implement and a set of negative features $\bar{F}$ it should not implement. The set of modules needed is then $M = featuresToModules(F, \bar{F})$.

- *Missing modules* whose implementing artifacts have to be manually added. $missing(M, A) = \{m \mid m \in M \wedge \nexists a \in A : m \in a|_{Modules} \vee (a|_{Modules} = \emptyset \wedge m \in a|_{Max})\}$
- *Surplus modules* whose implementing artifacts have to be manually removed. $surplus(M, A) = \{m \mid \exists a \in A : M \cap a|_{Modules} \neq \emptyset \wedge m \in a|_{Modules} \wedge m \notin M\}$
- *Uncertain order* of artifacts whose order has to be manually set. This is the case whenever children n_1 and n_2 of an ordered node N are put into a product together whose order is not determined by the sequence graph in N, i.e. $(n_1|_{SequenceNumber}, n_2|_{SequenceNumber}) \notin N|_{SequenceGraph}|_{Order} \wedge (n_2|_{SequenceNumber}, n_1|_{SequenceNumber}) \notin N|_{SequenceGraph}|_{Order}$.

5.2 Dependencies

In addition to the extracted traces the composition can make use of the dependencies between artifacts as well as the extracted dependencies between associations.

The composition has the following options when including an artifact that references another artifact that is, according to the extracted traces, not to be included.

- Leave reference unresolved (this option does only apply to *Cross-Tree* dependencies).
- Also include referenced artifact (recursively).
- Do not include referencing artifact (recursively).
- Include all artifacts in the same association as the referenced artifact (recursively).

Using this information the composition can potentially reduce the effort for manually completing a product.
Example: in the case of surplus warnings originating from an association A with $A|_{Modules} = \{\delta^1(\texttt{line}, \texttt{wipe}), \texttt{wipe}\}$ where only the base module `wipe` is needed and the derivative module $\delta^1(\texttt{line}, \texttt{wipe})$ is surplus, meaning another association A_2 containg base module `line` is not included. Those artifacts in A that reference artifacts in A_2 could then not be included in the composed product under the assumption that these are the artifacts that are responsible for the implementation of $\delta^1(\texttt{line}, \texttt{wipe})$, i.e. for the interaction of features line and wipe.

In addition the composition can use the extracted dependency information to provide a simple variability model (e.g. a simplified form of feature model) to help a software engineer configure a system instead of just a list of features. For example composing a draw application without the feature `BASE` selected which contains the drawing area would not make any sense, and that would be reflected in the model.

5.3 Tool

In [Fischer, 2013] a tool was developed making use of the presented extraction framework in the previous chapter and providing the composition functionality presented in this chapter. It provides a graphical user interface in the form of an Eclipse plugin. A screenshot is shown in Figure 5.1.

The tool also has the ability to convert Java source code into an Ecore representation that can later also be converted back to Java source code

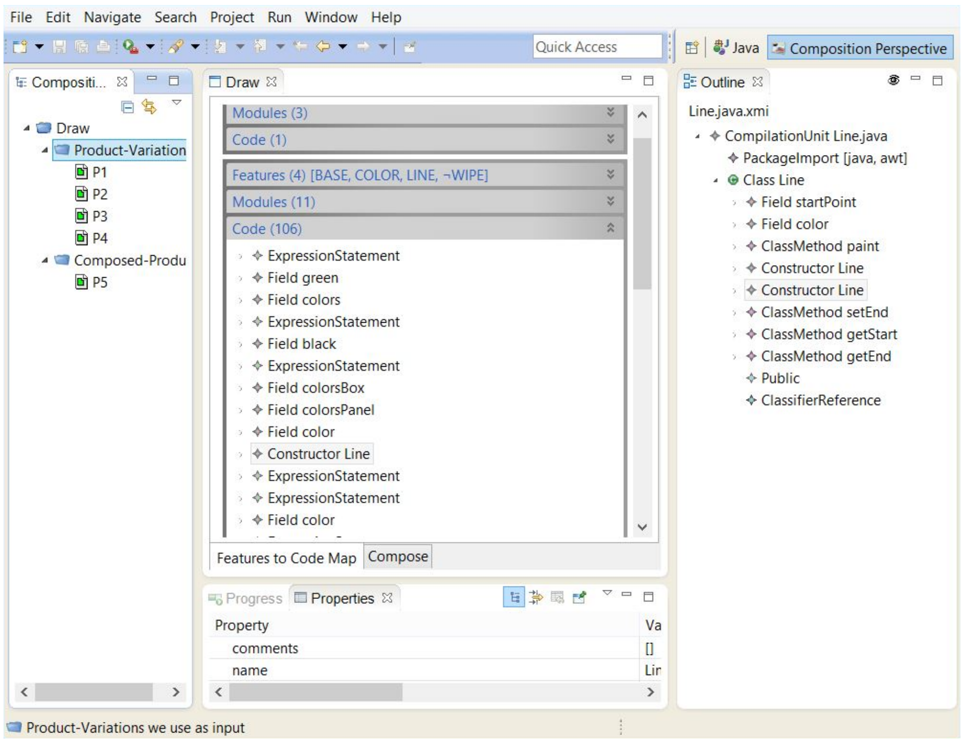
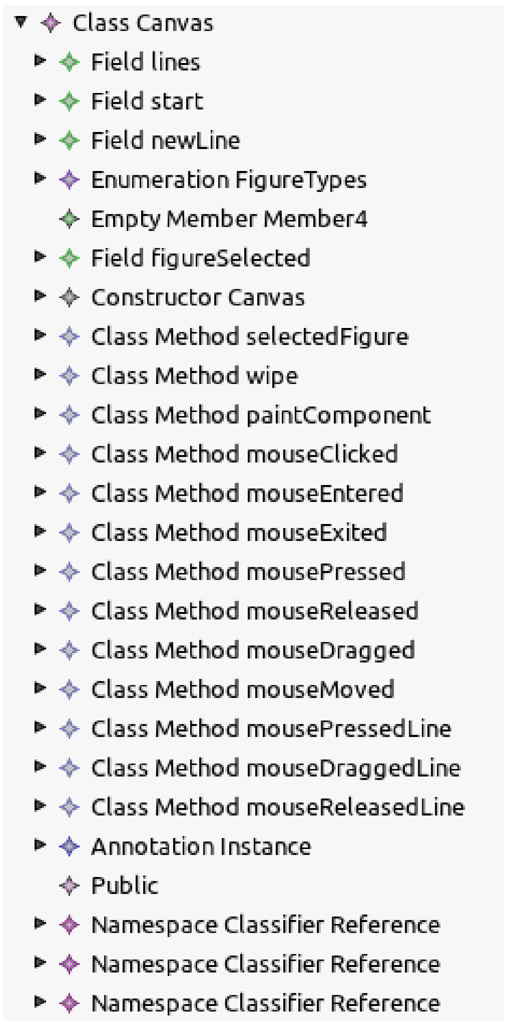


Figure 5.1: Composition Tool Screenshot

again. For the conversion the tool uses JaMoPP [JaMoPP, 2013]. The Ecore representation of the class `Canvas` is shown in Figure 5.2. Ecore models are tree structures that also allow for cross-references, which fits our chosen artifact trees perfectly.



```
▼ Class Canvas
    ▶ Field lines
    ▶ Field start
    ▶ Field newLine
    ▶ Enumeration FigureTypes
      Empty Member Member4
    ▶ Field figureSelected
    ▶ Constructor Canvas
    ▶ Class Method selectedFigure
    ▶ Class Method wipe
    ▶ Class Method paintComponent
    ▶ Class Method mouseClicked
    ▶ Class Method mouseEntered
    ▶ Class Method mouseExited
    ▶ Class Method mousePressed
    ▶ Class Method mouseReleased
    ▶ Class Method mouseDragged
    ▶ Class Method mouseMoved
    ▶ Class Method mousePressedLine
    ▶ Class Method mouseDraggedLine
    ▶ Class Method mouseReleasedLine
    ▶ Annotation Instance
      Public
    ▶ Namespace Classifier Reference
    ▶ Namespace Classifier Reference
    ▶ Namespace Classifier Reference
```

Figure 5.2: Ecore Tree of Class `Canvas` from Product $\texttt{P}_1$

Chapter 6

Approach

Based on the previously described *Extraction* and *Composition* [Fischer, 2013] we present an approach for developing and maintaining software product portfolios (together with [Fischer, 2013]). An overview of the proposed approach is shown in Figure 6.1. It starts with a set of related *Initial Products* ①, which could for instance be the three initial drawing applications used in our running example. These related products are used as initial input for the *Extraction* ② that is responsible for computing the associations that map modules to their implementation artifacts as well as determining the order of artifacts and extracting dependencies between associations. These extracted associations are stored in a *Database* ③. The *Composition* ④ can then use the information stored in the database to construct a product that is specified by the set of features it shall implement. However, it can not generate the artifacts responsible for the implementation of modules that have not been present in any of the input products used to build up the database, and it also can not separate artifacts whose modules have never appeared separately in any of the input products. Hence, the generated products can be incomplete. Therefore, in addition to the generated *Partial Product* ⑤, the approach also provides the software engineer with different types of *Warnings* ⑤ addressing the problems the product might have. These warnings guide the software engineer in making additional manual changes, if necessary, to finish the *Completed Product* ⑥. The completed product containing all the new information (like e.g. new module implementations, etc.) can then be used in the next iteration as additional input for the extraction to improve the quality of the automatically created partial products of future product variants by updating the database.

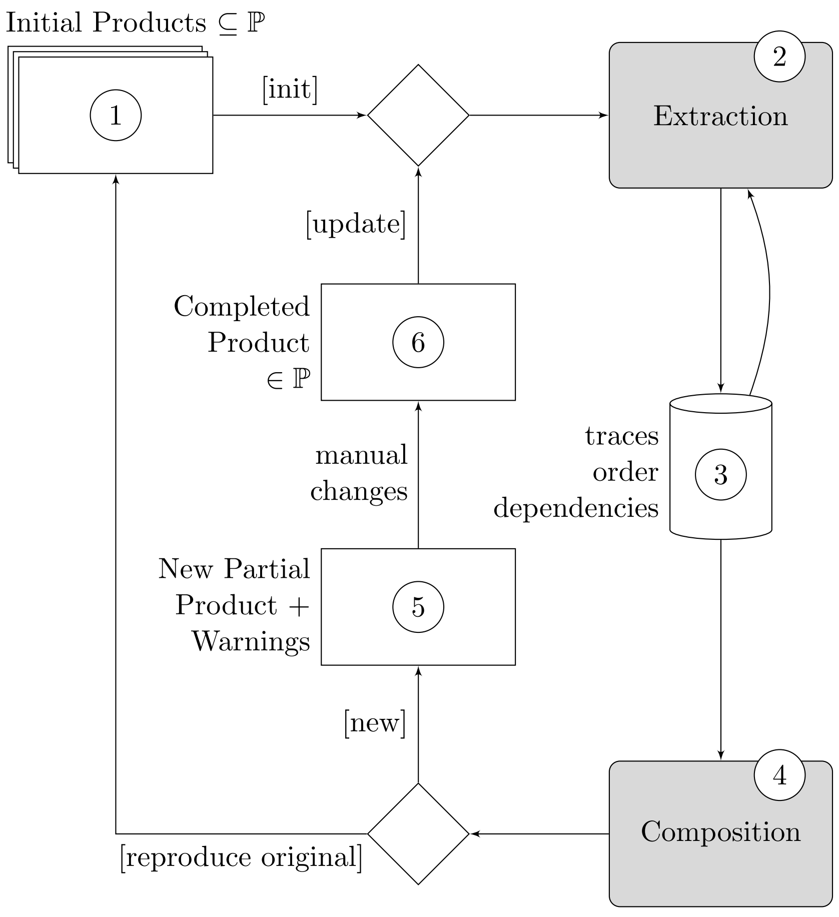


Figure 6.1: Framework Overview

The focus of this thesis is on the *Extraction.* A simple composition was implemented that is used for validation and evaluation purposes. A more sophisticated and advanced tool for the composition that makes use of the extraction framework presented in this thesis is developed in [Fischer, 2013]. The tool provides a graphical user interface in the form of an Eclipse plugin and allows for the conversion of Java source code to an Ecore representation.

The idea behind this approach is that software engineers can incrementally develop a product portfolio using the familiar and intuitive clone-and-own approach but without the problems that come with it in terms of maintenance and reuse management. The creation of the product portfolio should be similar to the clone-and-own principle while the maintenance of product variants and the reuse management when creating new product variants should be similar to single, configurable systems like SPLs.

There are essentially two ways that this approach can be used:

6.1 Incremental Way of Use

Software engineers can just start developing a product variant and add it to the database. Whenever a new variant is needed the composition can automatically create the initial skeleton. Reuse of existing implementation artifacts is then already taken care of. The software engineer can then, guided by the provided warnings, complete the product and again add it to the database to refine the extracted information and allow for better composition of future products. When changing a feature or making a bug fix this can be done at a central point directly on the database. All affected product variants (i.e. all those that require the changed module) can then simply be re-composed. These changes or fixes therefore do not have to be replicated for every variant.

Using this approach incrementally and consistently during the development and maintenance of product variants prevents them from diverging from each other which makes it easier to extract accurate information out of them.

6.2 Legacy Recovery

This applies when a company already has a portfolio of product variants for which it wishes to use this approach. The reason for this could be to better be able to maintain them and create new variants easier. The workflow remains unchanged. However, the extracted information may not be as accurate due to evolutionary changes in the product variants that may have diverged from each other over time. Therefore either a refactoring of the product variants before they are added to the database or a refactoring of the extracted informatin in the database after having added the legacy product variants should be performed. Therefore the upfront investment for recovering legacy variants is higher.

However, it might also make sense to do this for product variants of an SPL or any other single system representation of product variants. For example when new product variants are needed that an SPL was not designed for then the SPL could be transitioned to our approach by using its variants as input for the extraction. The new variants could then be composed using the composition. In such a scenario the problem of diverged product variants would not apply, because in an SPL this does not happen.

Chapter 7

Implementation

This chapter provides insight into the implementation of the *extraction framework* as a Java library. It is useful when intending to use the framework directly and for understanding the details of its internal workings. It can serve as a documentation and a handbook for the library.

The general structure of the framework is shown in Figure 7.1. The core is the *database*. Adding a product takes care of adding the features and converting it to an association. Alternatively it is also possible to add features and associations to the database directly. A database contains a set of associations (i.e. the already extracted traces) and a set of currently known features. Additionally the database provides statistics like runtime, number of added products and number of artifacts among others.

7.1 Features and Modules

Also shown in Figure 7.1 is the way features and modules are implemented.

A `Feature` is identified by its name, which is simply a Java String, that is used in `equals` and `hashCode` as an identifier. A `Negative Feature` inherits from `Feature`. A `Module` is then simply a set of features. As an implementation for sets we use the `HashSet` implementation from the Java class library. When checking if a feature is contained in a module it is a simple set operation. When comparing two modules then all contained features must match, which means the features' names must match. This results in a lot of set operations and a lot of string comparisons.

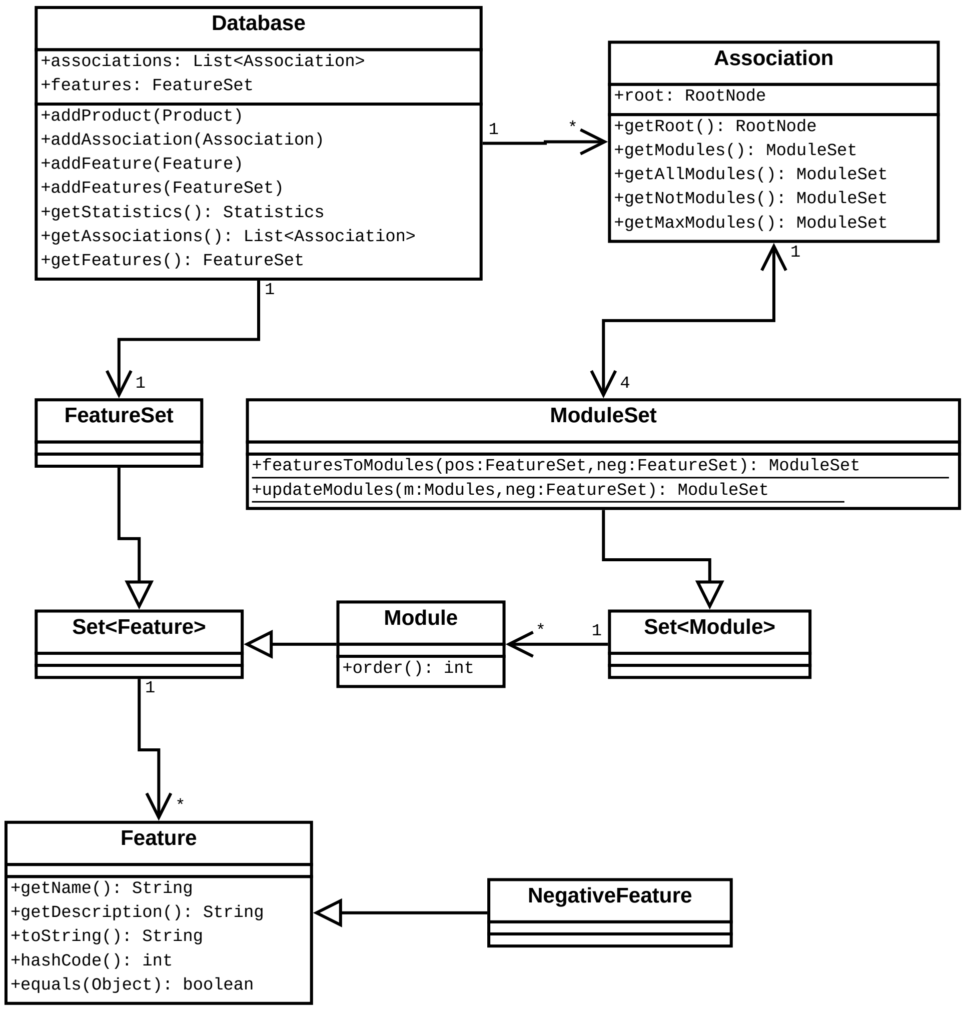

Database
+associations: List<Association>
+features: FeatureSet
+addProduct(Product)
+addAssociation(Association)
+addFeature(Feature)
+addFeatures(FeatureSet)
+getStatistics(): Statistics
+getAssociations(): List<Association>
+getFeatures(): FeatureSet
Association
+root: RootNode
+getRoot(): RootNode
+getModules(): ModuleSet
+getAllModules(): ModuleSet
+getNotModules(): ModuleSet
+getMaxModules(): ModuleSet
FeatureSet
ModuleSet
+featuresToModules(pos:FeatureSet,neg:FeatureSet): ModuleSet
+updateModules(m:Modules,neg:FeatureSet): ModuleSet
Set<Feature>
Module
+order(): int
Set<Module>
Feature
+getName(): String
+getDescription(): String
+toString(): String
+hashCode(): int
+equals(Object): boolean
NegativeFeature
1
*
1
1
1
4
*
1
1
*

Figure 7.1: UML Class Diagram for Database, Association, Model and Feature

Alternative Implementation

This subsection briefly describes an alternative and not so straight forward implementation, optimized for speed and memory consumption, that is not yet included in the framework.

Every feature f_i is not represented as a string anymore but as the number i with $0 \leq i < n$. The negative version of this features is represented as $-i$. A module m is then comprised of two n-bit numbers *pos* and *neg*: $m = (pos, neg)$. If a feature f_i is part of module m then the bit i in $m|_{pos}$ is 1, otherwise it is 0. If the negative version of a feature f_i is part of m then bit i in $m|_{neg}$ is 1, otherwise it is 0.

Feature f_i can then simply be added to m using bit operations. In Java: `m.pos = m.pos | (1 << i)` for positive features (similarly for negative features).

When checking if a positive feature f_i is contained in a module the following bit operations suffice: `(m.pos & (1 << i)) == (1 << i)`. Note that this does not have to be done separately for every feature, instead all desired features can be added to a module $features$ and it can be checked whether $features$ is a subset of m: `(m.pos & features.pos) == features.pos`. Alternatively, when composing a product with this set of features it can be checked whether module m should be included: `(m.pos & features.pos) == m.pos`. The same thing has to be done for the negative features.

Comparing two modules m_1 and m_2 now only involves the comparison of two n-bit numbers instead of set operations and string comparisons.

7.2 Nodes and Artifacts

The artifact and node structure is shown in Figure 7.2. The abstract base class `artifact` provides the general data structures for every artifact. We implemented two concrete kinds of artifacts:

- `EcoreArtifact` for Ecore models that can represent anything from UML diagrams to Java source code, and
- `JavaStringArtifact` which represents Java source code simply as Strings representing for example statements or method signatures.

The abstract base class `Node` again contains all the general data structures needed for nodes. There are three kinds of nodes:

- A `RootNode` is used as the root node for associations. It does not represent any implementation artifacts.
- An `UnorderedNode` is a simple unordered node without special functionality.
- An `OrderedNode` overwrites some of `Node`s methods that first perform some operations before calling the inherited super method. For example for an intersection first the alignment and sequencing is performed.

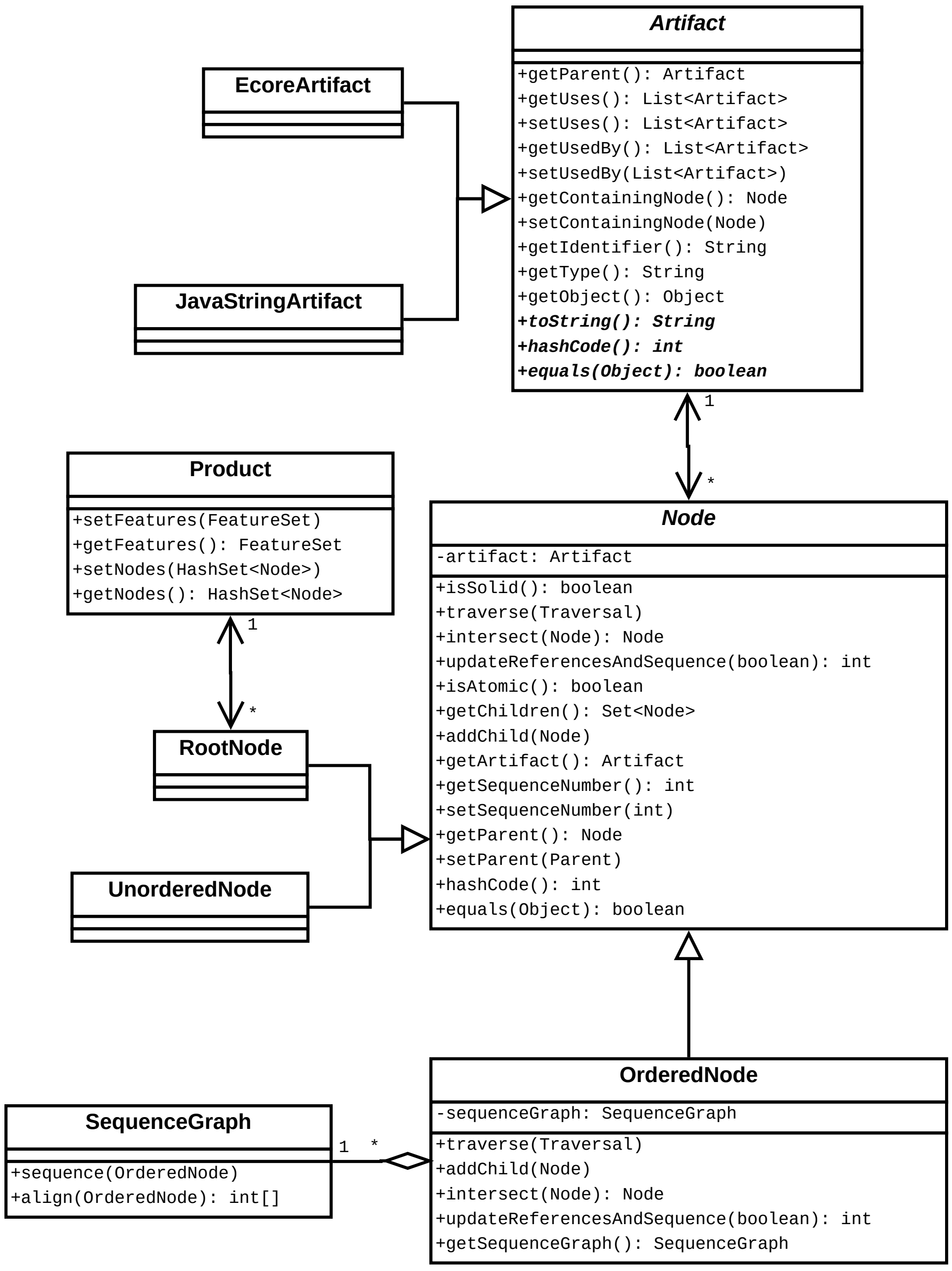


Figure 7.2: UML Class Diagram for Product, Artifact and Nodes

Note that all matching ordered nodes in the database (i.e. ordered nodes with the same sequence number, artifact and position in the artifact tree) have a reference to the same sequence graph and the same artifact, and this sequence graph and this artifact are referenced only by those nodes. Among such matching nodes exactly one is solid, all the others are weak. The extraction takes care that all the references between artifacts are always resolved correctly during the intersection and kept consistent within the database. This is important because for example during an intersection of two nodes, one already contained in the database and the other part of a new input product that is currently in the process of being added, only one artifact reference is kept, the other one is discarded.

7.3 Sequence Graph

The sequence graph is implemented as a directed acyclic graph structure where the edges (transitions) are labeled with sequence numbers. There is always exactly one root node and one leaf node. Every path from the root to the leaf represents a possible ordering. This implementation differs from the presented theory in Chapter 4. There the sequence graph was represented as a partial order relation. The order relation became larger the fewer ordering options there were. In an ideal case where an order for n artifacts is perfectly determined this would result in $n*(n-1)/2$ entries. The provided implementation instead shrinks the fewer ordering options there are for artifacts. The chosen representation for sequence graphs resembles *Labelled Transition Systems (LTS)*. The merging of two sequence graphs is therefore performed similarly to the *Parallel Composition of LTS*. Figure 7.3 shows an example. The sequence graph SG_2 (see Figure 7.3b) was aligned to sequence graph SG_1 (see Figure 7.3a) and the resulting sequence graph after merging these two is shown in Figure 7.3c. The order of statements with sequence numbers 2 and 7 is not determined. Therefore paths for both options exist in the graph.

Alternative Implementation

An alternative implementation could combine benefits of both the partial order relation and the graph representation by representing the order relation as a graph as shown in Chapter 4 in Figure 4.2c. The size of the graph would then be independent of the number of allowed ordering options. The difficulty might then be the merging step of two such partial order graphs.

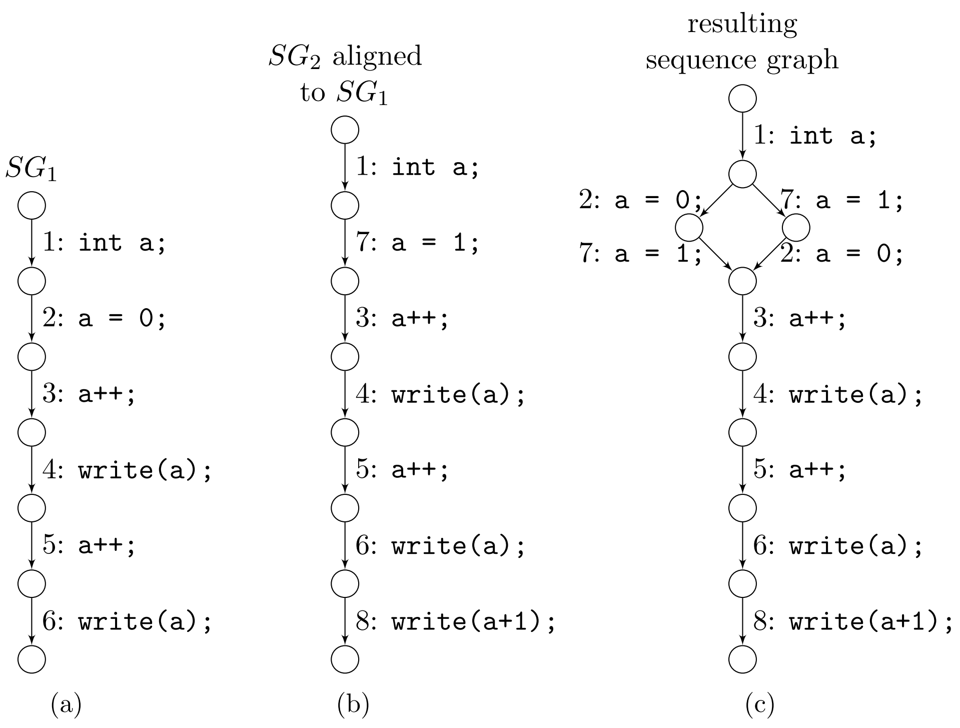


Figure 7.3: Sequencing of Ordered Artifacts

7.4 Traversals

Any additional functionality that is not inherent to the extraction framework is implemented in independent *traversals*. The base class `Traversal` provides basic functionality that is common to every traversal, like traversing a single `Node`, a complete `Product` or a full `Database`. Traversals can be used for implementing custom operations on the database and for extending the framework's functionality.

An overview about the currently implemented traversals is shown in Figure 7.4. They are described in more detail below.

- MergeTreesTraversal: Merges all artifact trees contained in a set of associations that are needed for composing a product with a given set of features.

- CheckConsistencyTraversal: Checks the database for consistency. For example a node N's parent must contain N as a child, or if an artifact A *uses* an artifact A' then A' must *be used* by A.

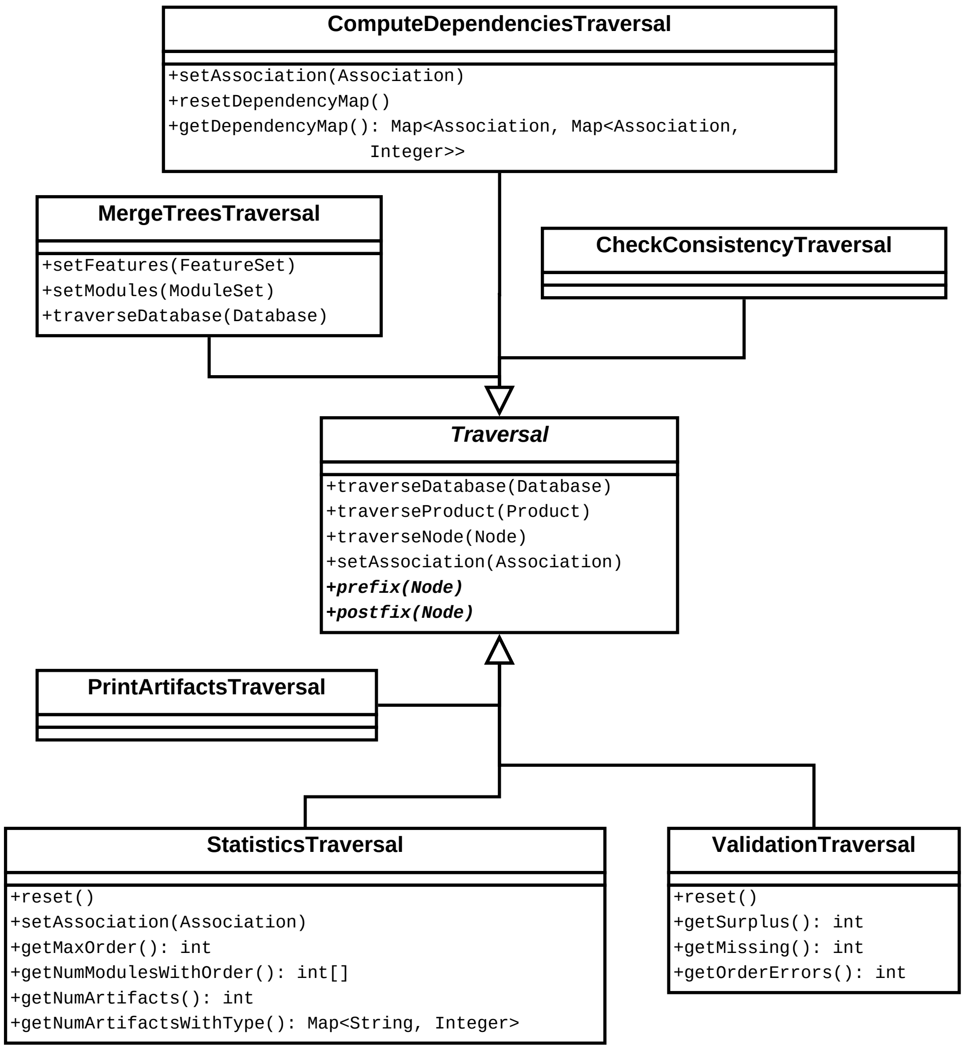

ComputeDependenciesTraversal
+setAssociation(Association)
+resetDependencyMap()
+getDependencyMap(): Map<Association, Map<Association, Integer>>
MergeTreesTraversal
+setFeatures(FeatureSet)
+setModules(ModuleSet)
+traverseDatabase(Database)
CheckConsistencyTraversal
Traversal
+traverseDatabase(Database)
+traverseProduct(Product)
+traverseNode(Node)
+setAssociation(Association)
+prefix(Node)
+postfix(Node)
PrintArtifactsTraversal
StatisticsTraversal
+reset()
+setAssociation(Association)
+getMaxOrder(): int
+getNumModulesWithOrder(): int[]
+getNumArtifacts(): int
+getNumArtifactsWithType(): Map<String, Integer>
ValidationTraversal
+reset()
+getSurplus(): int
+getMissing(): int
+getOrderErrors(): int

Figure 7.4: UML Class Diagram for Traversals

- StatisticsTraversal: Computes a number of metrics, for example the number of artifacts, the number of modules, etc.

- ComputeDependenciesTraversal: Computes the dependency graph for the current state of the database.

- ValidationTraversal: Compares an artifact tree with a reference artifact tree and computes the number of missing and surplus artifacts and the number of wrong orderings.

- PrintArtifactsTraversal: Simply prints an artifact tree to the console.

7.5 Parsers

A parser reads arbitrary objects and outputs an artifact tree to be used with our approach. Parsers for two kinds of objects are currently implemented:

7.5.1 Java Parser

The Java parser uses the Java Compiler API [JCAPI, 2013] to parse Java source code files and generates an artifact tree consisting of simple Java String artifacts. It does not extract dependencies between artifacts.

There are three types of this parser:

- Methods and Fields: Only creates artifact trees down to the level of methods and fields. This was used to compare the results with our previous work in [Linsbauer et al., 2013].
- Statements: Creates an artifact tree down to statement level whereas blocks (e.g. while-loops, for-loops, etc.) are treated as ordered nodes with the contained statements being their child nodes. This parser was used in the evaluation when computing the extraction and composition metrics (see next Chapter 8).
- Flat Statements: Also creates an artifact tree down to statement level but does not treat blocks as ordered nodes. Blocks do not have child nodes, instead they are simply followed by the nodes representing their contained statements.

7.5.2 Ecore Parser

This parser is used to parse Ecore models that adhere to the Java metamodel provided by [JaMoPP, 2013]. It creates an artifact tree that goes even below the level of statements. Statements are also broken up into their bits and pieces. For example every parameter of a method call is represented by a separate node. This parser extracts dependencies between artifacts and therefore was used when computing dependency graphs (see next Chapter 8).

Chapter 8

Evaluation

This section evaluates the proposed approach using 4 different case studies. First the case studies are introduced, then the extracted dependency graphs are shown and the metrics for the extraction followed by metrics for the composition are explained including the results for each of the case studies. The chapter concludes with a short analysis of the results.

8.1 Case Studies

The evaluation was performed using 4 case studies. An overview is shown in Table 8.1. The following subsections explain them in detail. All of these case studies are implemented in Java.

Case-Study	**#F**	**#P**	**LoC**	**#Art**
Draw	5	12	287 - 473	491
VOD	11	32	4.7K - 5.2K	5.5K+
ArgoUML	11	256	264K - 344K	192K+
ModelAnalyzer	13	5	35K - 59K	94K+

#F: Number of Features, #P: Number of Products, LoC: Range of Lines of Code, #Art: Number of Distinct Artifacts

Table 8.1: Case Studies Overview

The first three case studies, due to the fact that their product variants were generated from SPLs, represent quite ideal cases without any evolutionary changes where the product variants have not diverged from each other at all. This is also what would be expected when the presented approach is used incrementally from the very beginning or when the variants have been maintained exceptionally well.

The last case study *ModelAnalyzer* on the other hand represents the practical worst case scenario. Its variants have been developed independently from each other by different engineers over the course of many years and without ever reconciling any source code. Therefore its variants have diverged a lot from each other. For example some bug fixes have not been applied to all variants, others have been applied but by different engineers for each variant, each with their own coding styles. Additionally there are only 5 variants available for this case study. All this makes it extremely difficult for the extraction to achieve good results. This is what would be expected when recovering legacy product variants that have not been maintained very well to be used with our approach.

8.1.1 Draw

The *Draw* case study is an SPL of simple drawing applications of which some were used for illustration in previous chapters. It supports 5 features. Its feature model is shown in Figure 8.1. In total 12 product variants can be generated.

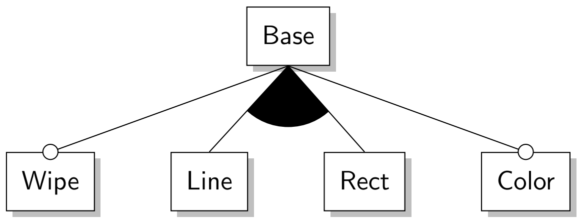


Figure 8.1: Feature Model for the Draw Case Study

8.1.2 VOD

VOD is an SPL for video-on-demand streaming applications. It supports 11 features of which 6 appear in every variant. The feature model is shown in Figure 8.2. It allows for the generation of 32 product variants.

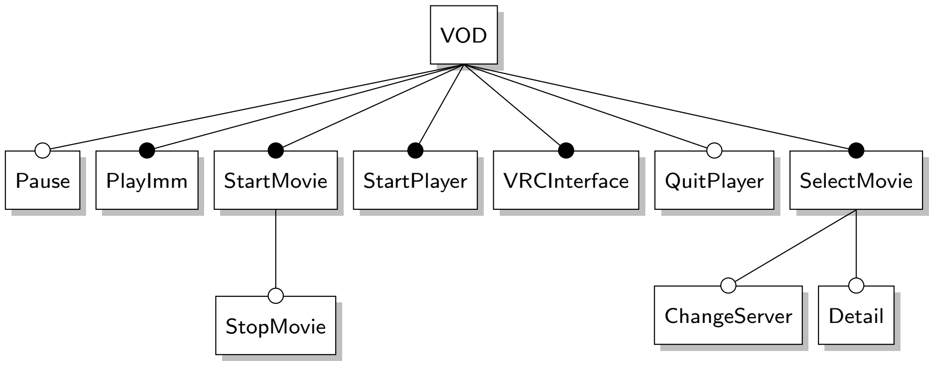


Figure 8.2: VOD Feature Model

8.1.3 ArgoUML

The largest case study *ArgoUML* is an open source UML modeling tool that was refactored into an SPL [Couto et al., 2011, ArgoUML, 2013]. It has 11 features of which 3 appear in every product variant. According to its feature model shown in Figure 8.3 there are 256 product variants.

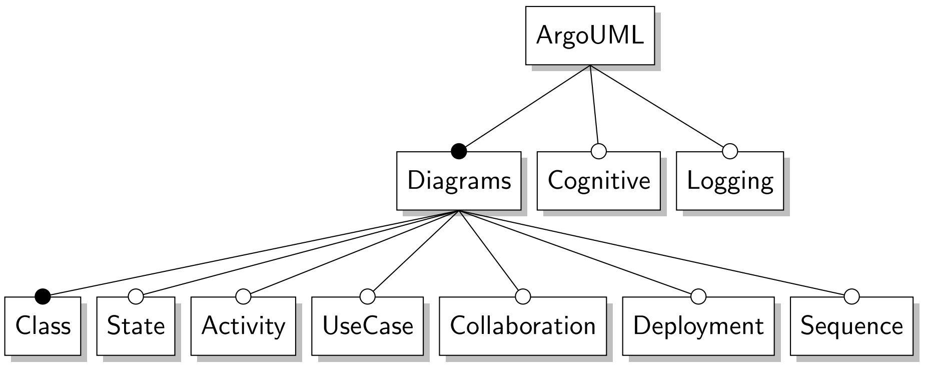


Figure 8.3: ArgoUML Feature Model

8.1.4 ModelAnalyzer

The last case study *ModelAnalyzer* is a consistency checking and repair technology. It is not an SPL, but rather its variants were created through copying from existing variants and then developed independently of each other by different engineers who each had their own goals. In total we had 5 different variants available with 13 features alltogether. Since ModelAnalyzer is not an SPL there is no feature model available. It is unknown how many possible variants there would be and what features are mandatory or optional.

The information about the variants (i.e. their source code and the features they implement) were obtained through interviews with the respective developers. Difficulties were for example that some developers had partially implemented features from other variants they copied from in their code and just left them in their unfinished state because they did not use that feature anyway. Also common names for features had to be established because different developers used different names for the same features. This is only to emphasize how difficult a case study ModelAnalyzer represents for the extraction process. Note that no prior cleanup of the variants was performed at all.

8.2 Dependency Graphs

This section shows the extracted dependency graphs for the Draw and VOD case studies. Dependencies between artifacts are only extracted by our Ecore Parser (see Chapter 7). These two case studies are the only ones for which an Ecore model representation was available in addition to their Java source code. The Java source code was converted into Ecore models using JaMoPP [JaMoPP, 2013]. Unfortunately this could not be done for the remaining case studies due to their size. The extracted dependency graphs are compared to the corresponding feature models to assess whether they are useful as simple variability models.

The shown dependency graphs are labeled with the corresponding association's lowest order modules. Base modules are depicted as solid boxes while derivative modules are depicted as dashed boxes. For better readability the self-dependencies are not shown, but they were always by far the strongest which could be a good indication for the extracted traces being correct.

Figure 8.4 shows the dependency graph for the Draw case study. Aside from the self-dependencies the strongest dependencies are towards the association containing the base module `base` which corresponds to feature `BASE` which in turn is the root node of the Draw feature model. The most and the strongest dependencies originate from the other associations containing base modules. When only considering these associations the graph resembles very much the Draw feature model in Figure 8.1.

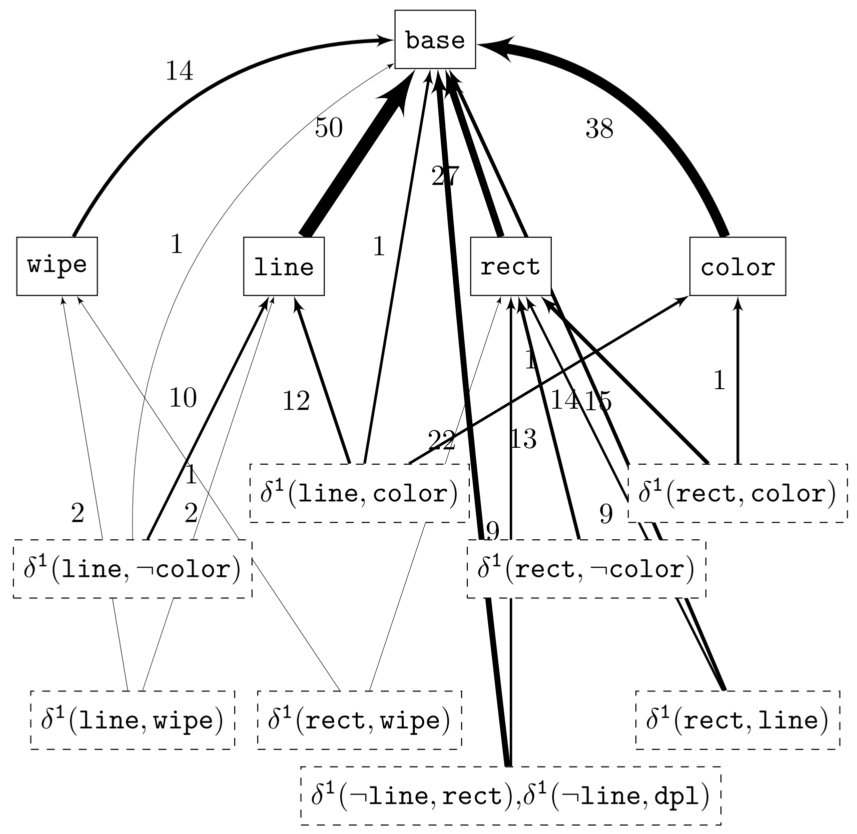
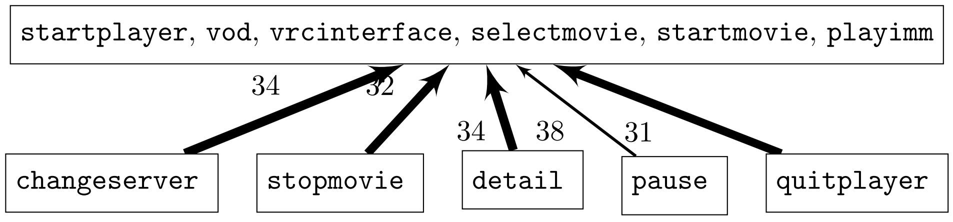


Figure 8.4: Dependency Graph for Draw Case Study

In Figure 8.5 the dependency graph for the VOD case study is shown. It is much simpler than the Draw graph. Since for VOD 6 features are always present in every variant they all appear in one association which is also the one with the strongest dependencies. All the other associations depend on it. In terms of modules and the corresponding features this again is reflected in the feature model in Figure 8.2.



startplayer, vod, vrcinterface, selectmovie, startmovie, playimm
34
32
34
38
31
changeserver
stopmovie
detail
pause
quitplayer

Figure 8.5: Dependency Graph for VOD Case Study

8.3 Extraction Metrics

The metrics presented in this section are used to assess the extraction. Their measurement or computation is performed during the creation of the database (i.e. during the extraction process) after each newly added product and/or on the final database once its generation is completed. Some of these metrics are computed using the `StatisticsTraversal` introduced in Chapter 7. Since the metrics depend not only on the number of input products but also on the features they implement, for each of the case studies 10 runs were performed, each using all the available product variants as input, but with a random order of the input products. The metrics are computed for every run and then averaged.

For computing these metrics the Java source code of the case studies was used and parsed using our Java parser (see Chapter 7).

8.3.1 Runtimes

We measured the runtimes of different portions of the extraction process, not including the parsing of the input product variants. Figure 8.6 shows the runtimes after each newly added product. As expected, adding another product takes longer the larger the database already is.

In Figure 8.7 the total runtimes for creating the complete database containing all available product variants are shown with respect to certain parts of the extraction process. $Runtime_{Products}$ is the execution time of the total extraction process. Additionally with $Runtime_{Associations}$ we measure the time for adding associations, which is a subset of $Runtime_{Products}$ not including the process of updating the database with new features. And lastly we also measure $Runtime_{Modules}$ and $Runtime_{Artifacts}$ for the portions of $Runtime_{Associations}$ in which the modules and the artifacts are compared respectively.

The most runtime is spent on processing the modules. The processing of the artifact trees only takes up a very small portion of the total runtime. Since the processing of modules is the least optimized code in the current implementation of the framework this is not troubling. It tells us that it is worth to spend time making optimizations to that part of the implementation as suggested in Section 7.1.

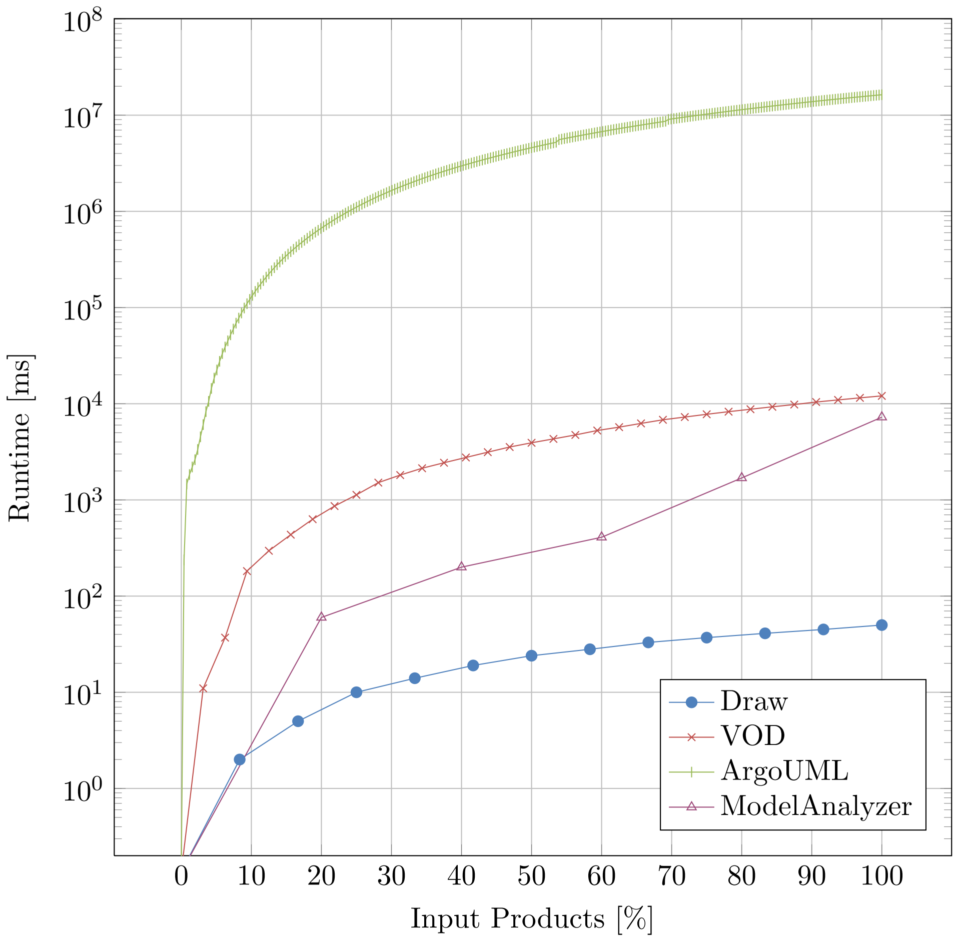


Figure 8.6: Runtime Overview

8.3.2 Modules per Order

The order of a module is a measure for the number of interacting features. A module of order o represents $o + 1$ interacting features. For example module $\delta^3(\texttt{base}, \texttt{line}, \texttt{wipe}, \texttt{color})$ is of order 3 which means it represents the interaction of 4 features. For every order we compute the total number of modules with that order that are associated with at least one artifact in the final database. This metric is interesting because it tells us up to what order modules actually require artifacts to implement them. Only modules in $A|_{Modules}$ for every association A in the database are considered here.

Given a number of features in a domain n the highest order derivative that can appear in that domain is $n - 1$. However, except for the ModelAn-

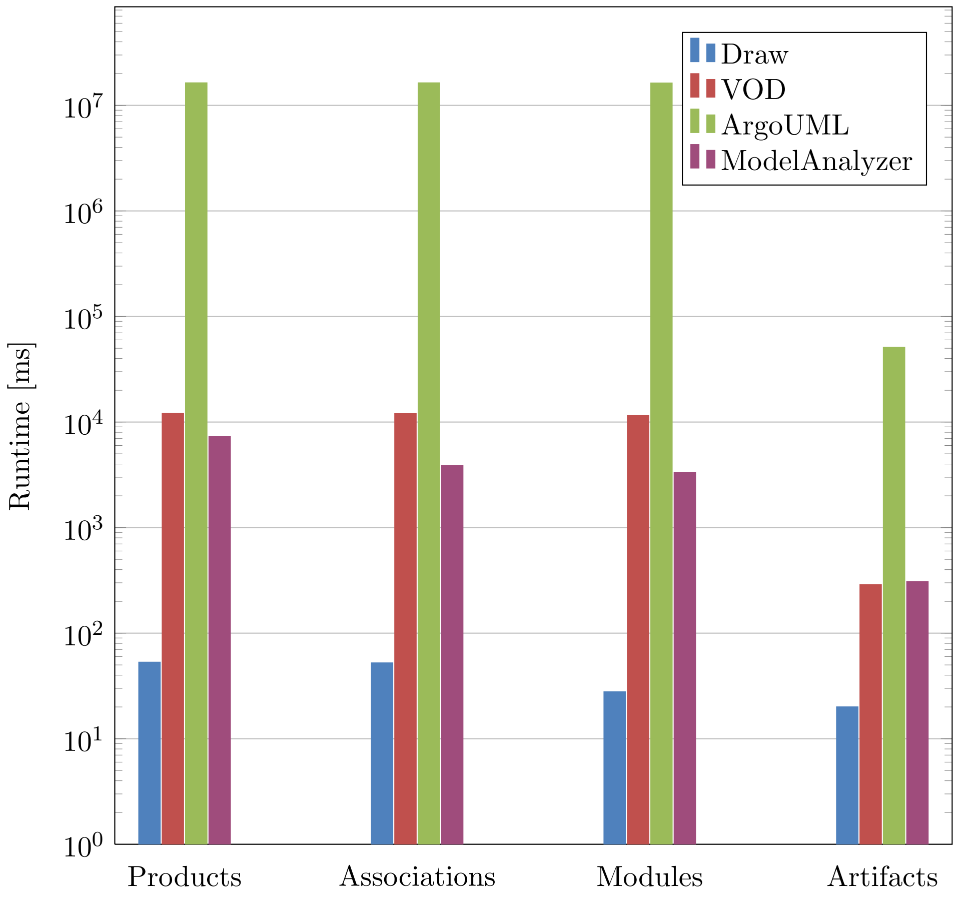


Figure 8.7: Runtimes Overview

alyzer case study none of the highest order derivative modules actually were associated with any artifacts. And for ModelAnalyzer that is only due to the fact that the number of available input products is very small and therefore the extraction could not rule out the possibility of some of the higher order derivatives containing code. Considering that it is increasingly unlikely for higher order derivatives to be associated with artifacts a *threshold* for the maximum order of derivatives could be used. This would reduce the number of modules and hence also reduce the number of generated warnings in the composition process and decrease the runtime, which, as we saw in the previous section, is mostly spent on processing modules.

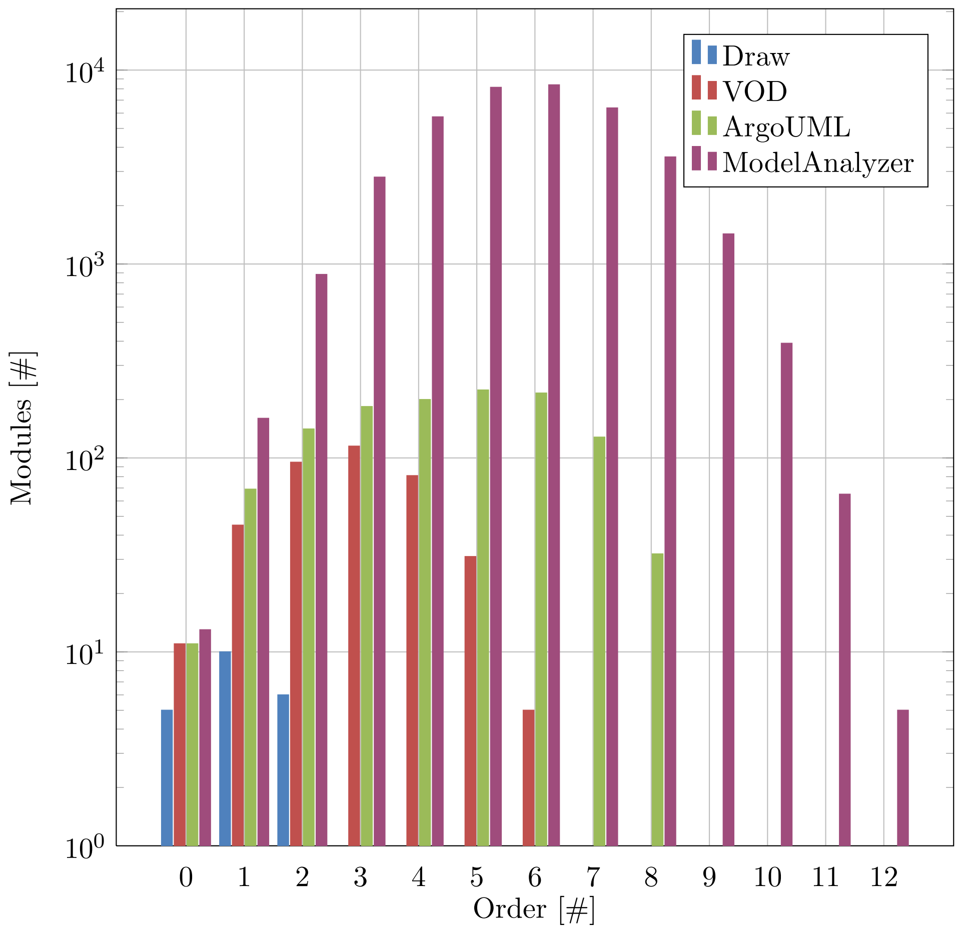


Figure 8.8: Modules per Order Overview

8.3.3 Artifacts

The number of artifacts in the database is a simple metric that hints at the size of the database. Figure 8.9 shows the number of artifacts after each newly added product. In every case study it takes only very few input products (less than 5) to have almost all the artifacts available. Further products improve other metrics like the number of associations or the distinguishability (see the following metrics) but do not improve much on the number of available artifacts.

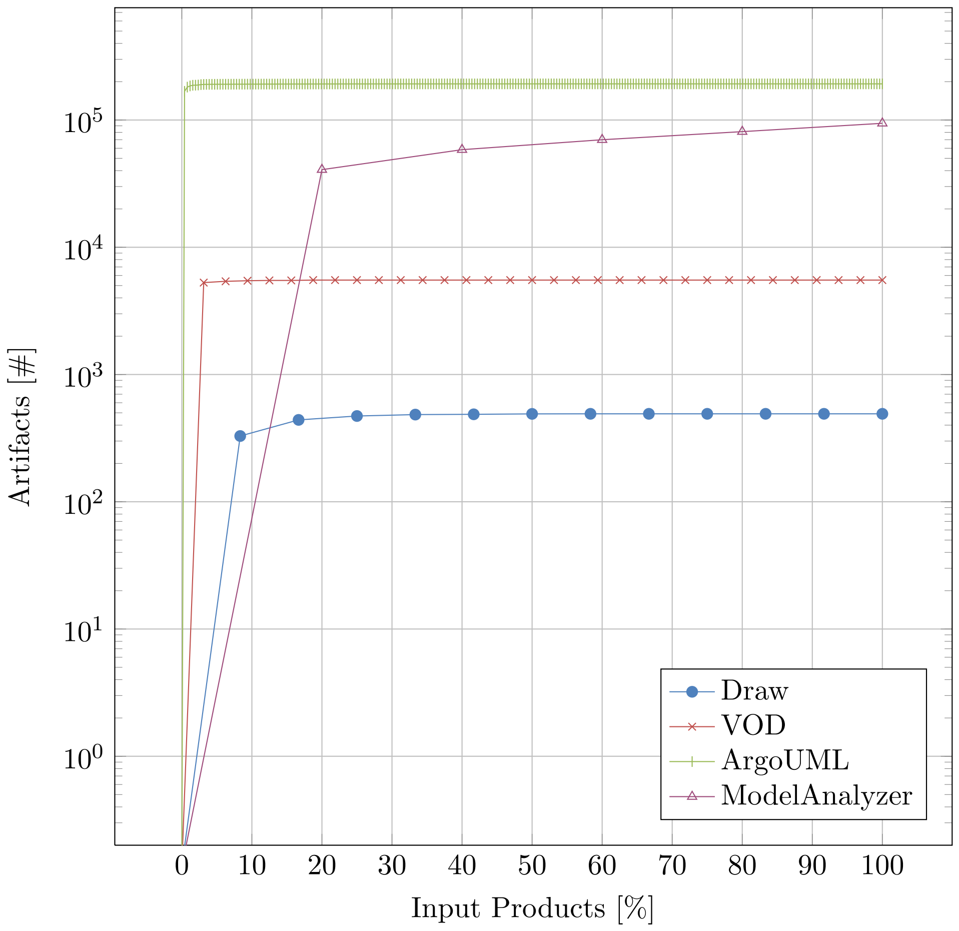


Figure 8.9: Artifacts Overview

8.3.4 Associations

The number of associations after each newly added product is shown in Figure 8.10. Similarly as for the number of artifacts also the number of associations increases very quickly with the first few input products already, although not with quite as few. However, it keeps increasing steadily with more additional products before it finally reaches its peak.

8.3.5 Distinguishability

Distinguishability describes the number of modules per association.

Definition 22 *Distinguishability is the average cardinality of all module sets whose respective associations contain at least one artifact and at least*

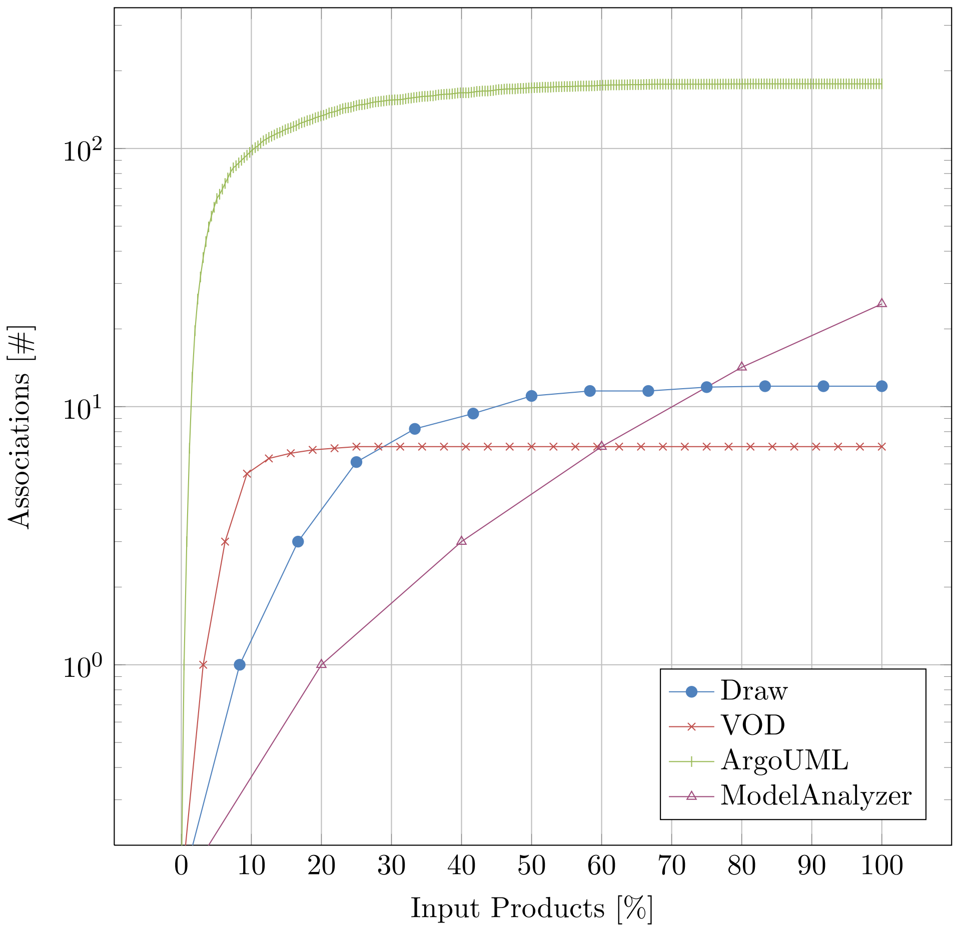


Figure 8.10: Associations Overview

one module.

$$Distinguishability = \frac{1}{n} * \sum_{i=1}^{n} |association_i|_{Modules}|$$

where n is the number of associations that contain at least one artifact and at least one module and $association_i$ is such an association.

This metric basically measures how many modules on average could not be separated because they never appeared without each other in any of the input product variants. The optimal value for this metric would be 1, meaning every association containing at least one artifact would have exactly one module. However, this can only very rarely be achieved due to mandatory features that can never appear without each other and are present in every product variant. In Table 8.2 the best reachable distinguishability and the best actually achieved distinguishability is shown for every case study except for ModelAnalyzer because there is no feature model available for it. The *distinguishability* improves with the number of input products and is generally worse the more features there are. As is shown in Figure 8.11 the distinguishability first gets worse quickly but then improves steadily with every additional input product. This is in contrast to every other metric so far. Only the ModelAnalyzer case study does not reach the point where the distinguishability improves because of the small number of available product variants.

Case-Study	**Best Reachable**	**Best Achieved**
Draw	$2^1 - 1 = 1$	1.9
VOD	$2^6 - 1 = 63$	63.8
ArgoUML	$2^3 - 1 = 7$	7.9

Table 8.2: Distinguishability Overview

8.4 Composition Metrics

These metrics are used to describe the quality of composed products using the previously extracted information. The better the extracted informationt the better the composition will work.

As an initial run the evaluation uses all the existing product variants in a product portfolio to create the database, recomposes all of them and computes the composition metrics to show that the extracted information is sufficient to fully re-engineer all input products. Subsequently the number of product variants used as input for the creation of the database is decreased from 100% to 0% in steps by removing products randomly. Next we generate the products that were not among the input products using the composition procedure and compare them to the corresponding original product (i.e. the product with the same set of features) of the portfolio. This allows us to draw a conclusion on the quality of newly generated products that were not used as input. Again, the quality of the composed products depends not only on the number of input product variants, but also on the features they implement. Therefore for every decreasing step we perform 10 runs for the current number of input products, where we pick the input products

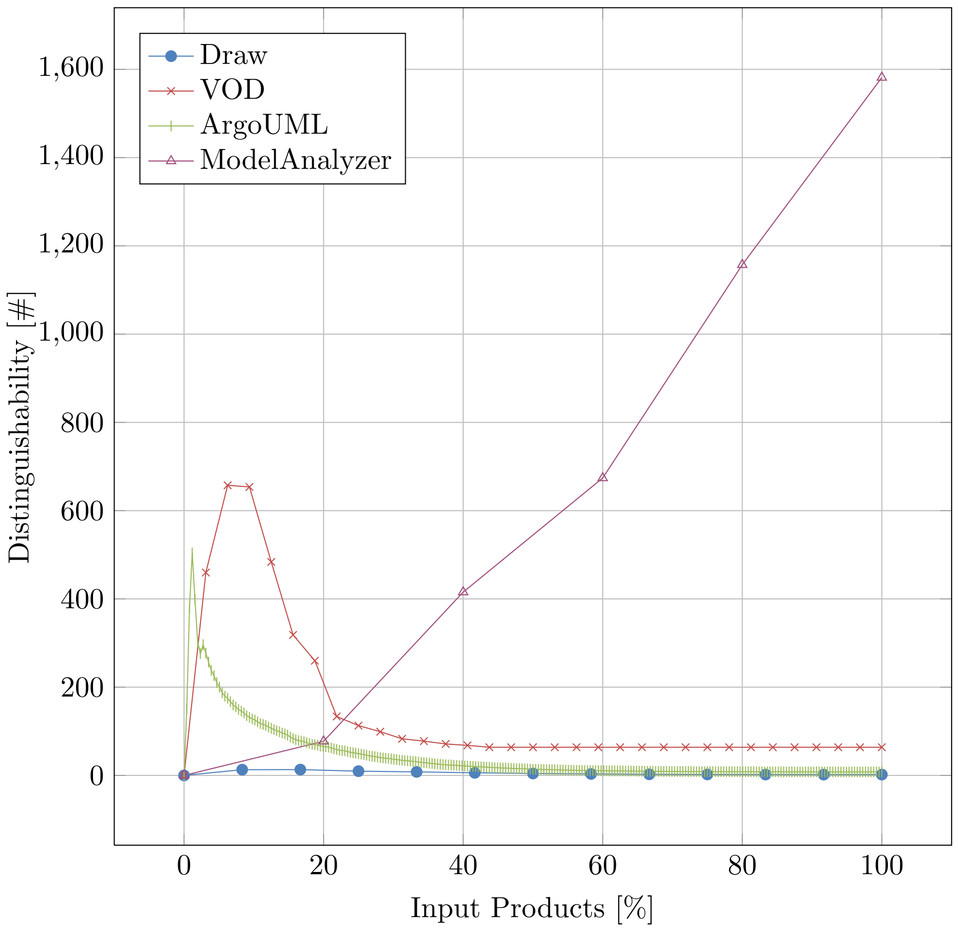


Figure 8.11: Distinguishability Overview

randomly. We compute the composition metrics for every composed product and then average them over all composed products of all runs with the same number of input products.

For computing these metrics the Java source code of the case studies was used and parsed using our Java parser (see Chapter 7). During the evaluation the composition was configured to include all structural dependencies (i.e. include a parent if at least one of its children is to be included even if the parent is not part of any of the selected associations) and to leave cross-tree dependencies between artifacts unresolved. The order of artifacts was considered as follows: if the correct order of artifacts was among the allowed orders in the corresponding node's sequence graph it was considered as correct, which was always the case in all the case studies.

8.4.1 Correctness

Definition 23 Correctness *is the percentage of overlap in artifacts between a composed product* $\mathtt{P}'$ *using the extracted information and the corresponding original product* $\mathtt{P}$ *(correctly aligned and sequenced to the database), where we consider only artifacts* N *stored in the database.*

$$Correctness = \frac{\|N \cap \mathtt{P}|_{RootNode} \cap \mathtt{P}'|_{RootNode}\|}{\|(N \cap \mathtt{P}|_{RootNode}) \cup \mathtt{P}'|_{RootNode}\|}$$

where $N = \bigcup_{as \in AS} as|_{RootNode}$ *with AS being the set of associations in the database.*

We use the correctness to express the quality of the composed products. It is affected negatively by surplus artifacts and by missing artifacts that would have been known to the composition (i.e. contained in the database), it is not affected however by artifacts that were not part of any of the input products and hence are not contained in the database.

The correctness for each of the used case studies with respect to the number of used input products is shown in Figure 8.12. Correctness increases quickly with the number of input products and reaches a near-optimal value very quickly at already about 15% of the available products used as input for the first three case studies. For the ModelAnalyzer it also increases quickly but then drops again. This is due to the fact that the variants only share a relatively small portion of their implementation artifacts. And the remaining large parts can not be separated, which is why with every new product a lot of surplus is introduced that outweighs the small number of additional artifacts that become available and are actually needed in the composed products. What makes it even more difficult is the small number of product variants available for ModelAnalyzer. Still the correctness lies between 40-60%.

8.4.2 Completeness

Definition 24 Completeness *is the percentage of artifacts from an original product* $\mathtt{P}$ *(correctly aligned and sequenced to the database) also found in the corresponding composed product* $\mathtt{P}'$*.*

$$Completeness = \frac{\|\mathtt{P}|_{RootNode} \cap \mathtt{P}'|_{RootNode}\|}{\|\mathtt{P}|_{RootNode}\|}$$

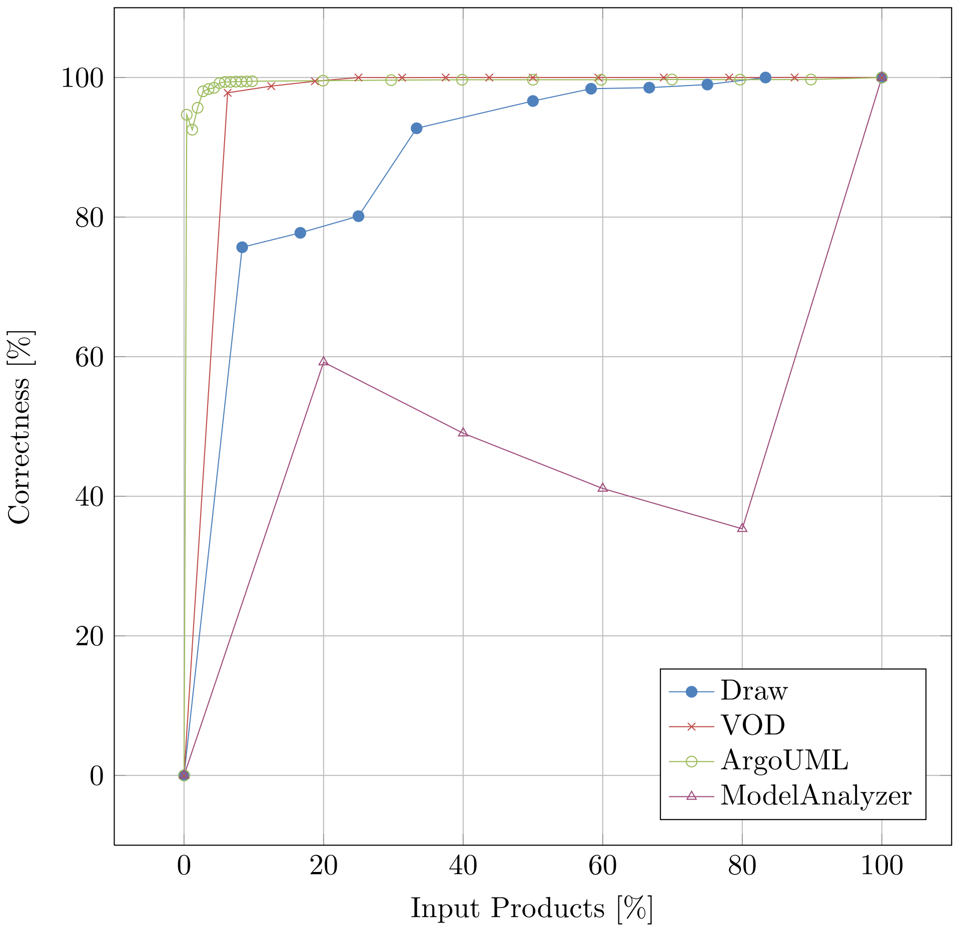


Figure 8.12: Correctness Overview

Completeness does not take surplus into account like the correctness does. It just tells us how much of what should be in a composed product is actually there, i.e. how much of its implementation could be automatically composed.

Figure 8.13 shows the completeness for all four case studies. The results look similar to the correctness. Similarly it reaches a near-optimum early already at roughly 15% of the available products as input. The results for the ModelAnalyzer are again not quite as good, but with values between 60% and 75% correctness still promising.

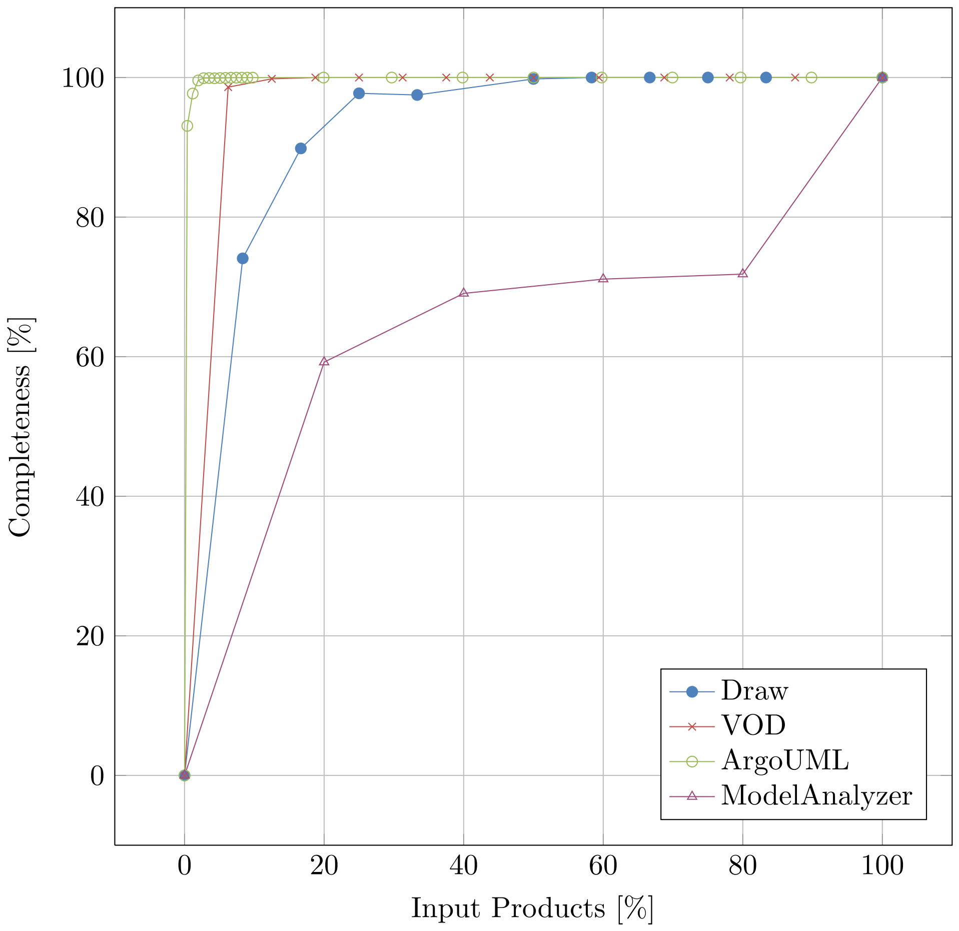


Figure 8.13: Completeness Overview

8.5 Analysis

In summary the results look very promising. The runtime for adding new products increases with the size of the database. The most runtime is taken up by processing modules, which is also the least optimized part of the framework's implementation and thus has a lot of potential for optimization. It makes sense to introduce a threshold for the maximum order of derivatives that will be computed to decrease the number of issued warnings and the runtime, because most higher order derivatives do not contain any implementation artifacts anyway. The number of artifacts in the database as well as the number extracted associations reach a peak already with very few input products, which means that the presented approach can already function well with just few available input products. Only to achieve a good distinguishability it is necessary to have a large set of available products.

What was learned from the extraction metrics also is reflected in the composition metrics. Composed products achieve high correctness and completeness with few input products already. Only the ModelAnalyzer case study did not achieve optimal, albeit still promising, results due to the highly diverged product variants in combination with the fact that only very few of these variants were available.

Chapter 9

Threats to Validity

The basic assumption of the presented extraction process is that the different product variants still have major portions of their implementation artifacts in common. However, if the variants have diverged too much during their development this assumption will not hold anymore. While there are ways to mitigate this (e.g. clone detection techniques) this can still be a challenge for the extraction. This problem can only arise if the approach is used at a later point during the development of the different product variants such that they have had enough time to diverge. If, however, the approach is applied from the very beginning the different variants will not get the chance to diverge and this threat will not apply. In this case the extraction will find almost ideal conditions and benefits from clone-and-own as well as from systematic reuse (e.g. SPLs) can be leveraged.

The current implementation of modules does not scale for large numbers of featurs. Fortunately, this is the least optimized part of the extraction and there is still a lot of optimization potential in this area. The computation of higher order derivatives that most likely are not associated with any implementation artifacts can be avoided by introducing a threshold for the order of derivative modules. Also, often the computation of all modules right off the start is not necessary, instead modules could be computed during the extraction process as needed, for example by using representative modules for a whole class of modules that is only split up when necessary. This way for example modules formed by features that never appear without each other never need to be computed and instead are simply represented by a single placeholder module.

Chapter 10

Related Work

In [Rubin et al., 2013] Rubin et al. present a set of formal operators as part of a framework for managing and refactoring product variants that were created through clone-and-own. They applied their operators on three industrial case studies and describe the activities that were carried out during the management of the product variants in the different case studies. We believe that our work can provide the functionalities of some of these operators.

In [Rubin and Chechik,] Rubin et al. survey different feature location techniques. Dit et al. also provide a survey on feature location techniques in [Dit et al., 2013]. The traceability extraction as part of our presented extraction approach can also be classified as such a technique.

In [Xue et al., 2012] Xue et al. discuss problems when using information retrieval techniques to identify features and their implementing code in a collection of product variants. To overcome these problems they exploit commonalities and differences of the product variants to improve the results achieved with information retrieval techniques.

In [Rubin and Chechik, 2012] Rubin et al. suggest two heuristics for improving the accuracy of feature location techniques when multiple product variants are available by comparing the code of a variant that contains a feature of interest to one that does not.

Chapter 11

Conclusions

We presented an extraction framework for extracting information out of sets of related product variants. This includes traceability information, possible orderings of artifacts and dependencies between traces. The evaluation showed that the extraction performs well with already a small number of input products and that all input products could always be reconstructed perfectly from the extracted information. We also found that the extracted dependency graphs could provide good approximations for simple variability models.

A brief overview was given about a composition tool that makes use of this extracted information to automatically compose new product variants and guide a software engineer in completing them.

Based on this extraction in combination with the composition we presented an approach for developing and maintaining a product portfolio as well as recovering legacy product variants for use with the presented approach.

Chapter 12

Future Work

Our future work includes but is not limited to the following:

- We want to evaluate the approach in more detail by means of more case studies, ideally using real world software systems from industry.
- To deal with evolutionary changes in product variants that have diverged from each other, we plan on applying clone detection techniques to make the approach more robust in such cases.
- The implementation of modules as well as the implementation of sequence graphs should be optimized.
- The extracted dependency graphs look promising as an approximation for simple variability models, we want to investigate on this further.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ArgoUML, 2013] ArgoUML (2013). Argouml-spl project. `http://argouml-spl.tigris.org/`. (accessed 2012).

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

[Couto et al., 2011] Couto, M. V., Valente, M. T., and Figueiredo, E. (2011). Extracting software product lines: A case study using conditional compilation. In *CSMR*, pages 191–200. IEEE Computer Society.

[Dit et al., 2013] Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

[Dubinsky et al., 2013] Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K. (2013). An exploratory study of cloning in industrial software product lines. In Cleve, A., Ricca, F., and Cerioli, M., editors, *CSMR*, pages 25–34. IEEE Computer Society.

[Fischer, 2013] Fischer, S. (to appear in 2013). Feature-based composition of software-systems. In *Master's Thesis, Johannes Kepler University Linz*.

[Haslinger et al., 2011] Haslinger, E. N., Lopez-Herrejon, R. E., and Egyed, A. (2011). Reverse engineering feature models from programs' feature sets. In Pinzger, M., Poshyvanyk, D., and Buckley, J., editors, *WCRE*, pages 308–312. IEEE Computer Society.

[JaMoPP, 2013] JaMoPP (2013). Jamopp. `http://www.jamopp.org/`. (accessed 2013).

[JCAPI, 2013] JCAPI (2013). Source code analysis using java
6 apis. `http://today.java.net/pub/a/today/2008/04/10/`

source-code-analysis-using-java-6-compiler-apis.html. (accessed 2012).

[Kishi et al., 2013] Kishi, T., Jarzabek, S., and Gnesi, S., editors (2013). *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*. ACM.

[Linsbauer et al., 2013] Linsbauer, L., Lopez-Herrejon, R. E., and Egyed, A. (2013). Recovering traceability between features and code in product variants. In [Kishi et al., 2013], pages 131–140.

[Liu et al., 2006] Liu, J., Batory, D., and Lengauer, C. (2006). Feature oriented refactoring of legacy applications. In *Proc. of 28th int. conf. on Software engineering*, ICSE '06, pages 112–121, New York, NY, USA. ACM.

[Metzger and Pohl, 2007] Metzger, A. and Pohl, K. (2007). Variability management in software product line engineering. In *ICSE Companion*, pages 186–187. IEEE Computer Society.

[Northrop, 2008] Northrop, L. (2008). Software product lines essentials. `http://www.sei.cmu.edu/library/assets/spl-essentials.pdf`. (accessed 2012).

[Rubin and Chechik,] Rubin, J. and Chechik, M. A survey of feature location techniques. *Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer, To appear.*

[Rubin and Chechik, 2012] Rubin, J. and Chechik, M. (2012). Locating distinguishing features using diff sets. In *ASE*, pages 242–245. ACM.

[Rubin et al., 2013] Rubin, J., Czarnecki, K., and Chechik, M. (2013). Managing cloned variants: a framework and experience. In [Kishi et al., 2013], pages 101–110.

[van d. Linden et al., 2007] van d. Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

[Xue et al., 2012] Xue, Y., Xing, Z., and Jarzabek, S. (2012). Feature location in a collection of product variants. In *WCRE*, pages 145–154. IEEE Computer Society.

Goal

Extracting information out of sets of product variants:

- Statement Level Traceability (ordering, ...)
- Dealing with Non-Unique Traces
- Generalization to other kinds of artifacts (languages or models as source code or e.g. Ecore)
- Dependencies (uses, used by, parent, children, ...)
- Allow adding new products incrementally.
- Providing a Java implementation of the framework in the form of a Java library and API.

Danksagung

Ich möchte Univ.-Prof. Dr. Alexander Egyed M. Sc. und Dr. Roberto Erick Lopez-Herrejon M. Sc. für die hervorragende Unterstützung und Betreuung danken die sie mir während der Erstellung dieser Arbeit und auch darüber hinaus zukommen haben lassen.

Besonderer Dank gilt auch meiner Familie die es mir während meines Studiums sehr leicht gemacht hat und mich unterstützt hat wo immer es ging.

Lebenslauf

Persönliche Daten

Name	Lukas Linsbauer
Geburtsdatum	1. März 1989
Staatsbürgerschaft	Österreich
Führerschein	B

Ausbildung

2013 - jetzt	Master Studium Informatik an der Johannes Kepler Universität Linz
2009 - 2012	Bachelor Studium Informatik an der Johannes Kepler Universität Linz
2008 - 2009	Zivildienst als Rettungssanitäter in Linz
2003 - 2008	HTL für EDV und Organisation Leonding

Berufserfahrung

2013 - jetzt	Studentischer Mitarbeiter am Institute for Systems Engineering and Automation an der JKU
2012	2 Monate Praktikum am Institute for Systems Engineering and Automation an der JKU
2010	Freier Dienstnehmer im IT Gewerbe
2009/2010	Angestellter bei Gumpinger Software
2008	Angestellter bei Gumpinger Software
2007/2008	Projektarbeit im Rahmen der HTL zur Digitalisierung der Schülerfreifahrtsanträge (digitale Signaturen) in Zusammenarbeit mit OÖVG und Gevas
2007	7 Wochen Praktikum bei der RACON Software GmbH
2006	6 Wochen Praktikum bei der RACON Software GmbH
2004	4 Wochen Praktikum bei der Werbeagentur adeins in Linz (Webentwicklung)

Publikationen

Lukas Linsbauer, Roberto E. Lopez-Herrejon, Alexander Egyed: *Recovering Traceability between Features and Code in Product Variants.* SPLC 2013: 131-140

Sonstiges

Leistungsstipendium der TNF JKU für Studienjahr 2009/2010.
Leistungsstipendium der TNF JKU für Studienjahr 2010/2011.
Leistungsstipendium der TNF JKU für Studienjahr 2011/2012.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, September 2013

Lukas Linsbauer