An Ontological Approach for Identifying Variants: The Cases of Specialization and Template Instantiation

Iris Reinhartz-Berger\textsuperscript{a}, Anna Zamansky\textsuperscript{a}, Yair Wand\textsuperscript{b}

\textsuperscript{a}Department of Information Systems, University of Haifa, Haifa 31905, Israel
\textsuperscript{b}Sauder School of Business, University of British Columbia, Vancouver, Canada

\{iris,annazam\}@is.haifa.ac.il, yair.wand@ubc.ca

ABSTRACT. Software is a crucial component in many products and often is a product in itself. Software artifacts are often developed for particular needs. However, identifying software variants is important for increasing reuse, reducing time and costs of development and maintenance, increasing quality and reliability, and improving productivity. We propose a method for utilizing variability mechanisms from Software Product Line Engineering (SPLE) for enabling identification of variants of software artifacts. The method is based on an ontological formal framework for representing variability. We demonstrate the feasibility of the method on two common variability mechanisms, specialization and template instantiation. The method has been implemented for utilizing reverse engineered code, providing a proof-of-concept of its feasibility.

Keywords: Variability, Reuse, Software Product Line Engineering

1 Introduction

As development becomes increasingly complex while reducing time-to-market remains a critical issue, identifying variants of software artifacts, such as requirements, design models, and code, plays a central role in software engineering. Variability is specifically researched and studied in the field of Software Product Line Engineering (SPLE) \cite{9, 24}, which aims at providing techniques, methods, and tools for effectively and efficiently developing and maintaining similar software products. This is done by promoting systematic reuse through what is commonly called variability mechanisms (a.k.a. reuse or variation mechanisms). These are techniques applied to adapt generic (reusable) artifacts to the context of particular products. Different variability mechanisms have been suggested over the years for different development stages, e.g., implementation \cite{2}, architecture design \cite{4}, and reference models \cite{5, 7}. Specialization (a.k.a. inheritance) and template instantiation are two examples of such mechanisms relevant throughout the whole development lifecycle. Specialization deals with refining behaviors (commonly by introducing new attributes or increasing...
the ranges of existing attributes) or adding behaviors. Unlike code inheritance, which sometimes “just” promotes software reuse or behavior substitution (through operation overriding), in specialization the specialized element is expected not to “violate” the laws (i.e., the intended behavior) of the generalized element, but to refine it. *Template instantiation* enables adapting types or filling product-specific parts in a generic behavior\(^1\).

Despite the benefits of reuse in general and SPLE in particular, in practice artifacts are often not developed for reuse. Cloning, for example, involves copying artifacts and adapting them to the particular needs and requirements. It provides a common, available, and simple practice [11], [16] whose consequences are identified in adaptation, bug fixing, and maintenance. Even when dealing with software product lines, a recent industrial survey [6] reveals that SPLE is commonly adopted extractively (i.e., existing product artifacts are re-engineered into a software product line) or reactively (i.e., one or several products are built before the product line artifacts are developed).

In those scenarios identification of variants is required to support adaptation and improve maintenance or bug fixing. However, existing approaches identify variants or clones, utilizing syntactic or semantic techniques, in order to refactor or track them to avoid inconsistencies [22]. Other works analyze the variability of certain types of software artifacts (most notably, code or requirements), e.g., [10], [14].

We propose a method for identifying variants of software artifacts and associating them with variability mechanisms to potentially increase their reuse and improve their future development and maintenance. A formal framework for representing properties of variability has already been presented in [26] and used to define different variability mechanisms. In that framework, using the ontological approach of Bunge [8], software products and software product lines are defined as things exhibiting behavior. The framework identifies relationships among software product lines and software products and enables mathematical definition of well-known variability mechanisms.

In this paper we suggest how to identify situations in software artifacts where different variability mechanisms may potentially be applied. The method is composed of three stages, as depicted in Fig. 1. In the first stage the information regarding the software products is extracted from their artifacts and represented in a repository. Next, the commonality and variability are analyzed utilizing properties of different variability mechanisms, such as specialization and template instantiation. Finally, the analysis results are presented in variability models, expressed in languages such as feature diagrams [15] or Orthogonal Variability Models (OVM) [24]. The actual application of the mechanisms is currently left to software designers and implementers.

Below, Section 2 reviews the relevant literature. Section 3 briefly presents the ontological foundations. Section 04 introduces the formal basis for the suggested method, while Section 5 presents its realization and refers to preliminary results. Finally, Section 6 concludes and sets the ground for future research.

---

\(^1\) Template instantiation is realized in Java and C#, for example, through the concept of generics.
2 Related Work

Identifying variants of software artifacts has been studied for several purposes. First, techniques have been suggested to detect code clones and manage them through refactoring or tracking [22][27]. Roy and Cordy [25] refer to four categories of clone detection approaches: textual, lexical, syntactic, and semantic. After being detected, the clones can be divided to those that should be refactored (i.e., merged into a single fragment) and those that should be tracked in order to improve management (e.g., to support consistent update of all clones). In the context of SPLE, Faust et al. [12] propose a method for migrating multiple instances of code units of a “successful” single information system to a software product line. The method is based on a two phase model: (1) grow, in which the code is copied and modified to implement additional similar functionality, and (2) prune, in which the different variants are merged to support easy percolation of changes. Mende et al. [20] further suggest a tool to support the maintenance of code developed following the grow-and-prune model. In order to identify similar functions that may be merged, token-based clone detection is used to detect pairs of functions sharing code. Then, textual similarity measures are utilized to lift sufficiently similar functions to the architectural level.

Detection of variants is also studied for analyzing variability. Ryssel et al. [29], for example, propose how to automatically identify variation points, namely, places where variability occurs, in function-block-based models. These are models that decompose the functionality of systems into components (function blocks). Yoshimura et al. [32] describe an approach to detect variability in a software product line from the change history of the software. Several studies offer methods for generating variability models from existing artifacts, e.g., [10], [14], [23], [31]. Overall, these studies concentrate on identification of particular types of artifacts (most notably, code or requirements) and creates feature diagrams or Orthogonal Variability Models (OVM) [24] for representing variability. Moreover, many approaches for managing cloned variants make assumptions on the project context or the application domain [28].

In this paper we suggest identifying variants utilizing properties of commonly used variability mechanisms, such as specialization and template instantiation. The study in [3] have already referred to variability mechanisms as techniques to guide customization or modification of existing components. Our approach formalizes such guid-
ance for the general case and demonstrates it on two variability mechanisms and object-oriented design or code artifacts.

3 The Ontological Foundations

Our approach is based on the ontological model of Bunge [8]. Ontologies are commonly used to represent real world phenomena, to organize information, and to reduce complexity. We chose Bunge’s ontology because it has been used in conceptual modeling of information systems analysis and design [30]. It can be therefore considered a natural candidate for defining software artifacts and providing the semantics of variability mechanisms. Although other, more expressive ontological models could be used, we show next how Bunge’s concepts can be used for defining variability mechanisms and analyzing the differences between them. Elaboration on Bunge’s ontological model in the context of software variability analysis can be found at [26].

3.1 Things, States, and Behaviors

The elementary unit in Bunge’s ontology is a thing, which possesses properties (intrinsic and mutual) and manifests behaviors. Properties are known to humans via attributes. A chosen set of attributes forms the state variables by which we model things. The values of state variables change in time, due to the occurrence of an event which triggers changes of state. Events can be external – caused by changes in other things, or internal – caused by the thing itself. From an external view, the behavior of things can be modeled by initial state of the thing before the behavior occurs ($S_1$), the sequence of external events triggering the behavior ($<e>$), and the final state of the thing after the behavior occurs ($S^*$).

Table 1. The relevant concepts from Bunge’s ontological model

<table>
<thead>
<tr>
<th>Concept</th>
<th>Formal definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>State variable</td>
<td>An expression of the form $x_i$, which has an associated set Range($x_i$) of values that can be assigned to it</td>
<td>Range (status) = {available, unavailable}; Range (borrowable?) = {yes, no}</td>
</tr>
<tr>
<td>State (of the thing)</td>
<td>A (potentially partial) assignment of values (from the associated ranges) at a given time to the state variables of the thing</td>
<td>$s'=$(available, yes); $s'=$(unavailable, yes)</td>
</tr>
<tr>
<td>External event</td>
<td>A (trigger that causes) change in the state of a thing as a result of an action of another thing</td>
<td>User borrows User returns</td>
</tr>
<tr>
<td>Behavior*</td>
<td>A triple $b=(S_1, &lt;e&gt;, S^<em>)$, $S_1$ and $S^</em>$ are the initial and final states. $&lt;e&gt;$ is a sequence of external events</td>
<td>Borrowing: ($s'$, &lt;user borrows&gt;, $s'$); Returning: ($s''$, &lt;user returns&gt;, $s''$)</td>
</tr>
<tr>
<td>Thing</td>
<td>Described by $T=(SV, E, B)$; $SV$ is the state variables of interest. $E$ is a set of external events of interest. $B$ is a set of allowed behaviors.</td>
<td>Book copy = ([status, borrowable?], [user borrows, user returns], (borrowing, returning))</td>
</tr>
</tbody>
</table>

* This concept is actually an extension to Bunge’s ontological model, introduced in [25].
Table 1 summarizes these concepts and exemplifies them using library management systems. Book Copy is a thing, characterized by status (available or unavailable) and an indicator whether it can be borrowed or not (due to library’s policies).

### 3.2 The Formal Framework for Representing Variability Mechanisms

Using Bunge’s ontology, software product lines and software products are represented by things exhibiting behaviors. Product artifacts are descriptions of software products and core assets are descriptions of software product lines which determine (part of) the behavior of software products. Product artifacts are obtained by introducing modifications to core assets.

Modifications can be classified along two dimensions: product and element. In the **product dimension** we examine the relationship between the whole set of behaviors \( B_P \) of a product (as specified in the relevant product artifact) and the whole set of the software product line behaviors \( B_{SPL} \) (as specified in the relevant core asset). However, in this paper we focus on the **element dimension** which deals with the relationship between a single behavior of a software product, \( b_P \), and the corresponding (single) behavior of the software product line - \( b_{SPL} \). The premise is that \( b_P \) “concretizes” \( b_{SPL} \). Concretization allows for changes in the use of state variables (from the core asset) and in their allowed values. This is described by a state mapping from the set of states for the software product line to the set of states for a software product. The state mapping is induced by the relevant state variable mapping and the value mapping (see [26] for the full definitions of these mappings).

Concretization of a behavior \( b_{SPL} = ( S_1, < e >, S^* ) \) can be achieved in different ways, two of which are specialized and template-instantiated behaviors (see Table 2).

<table>
<thead>
<tr>
<th>Variability of Behaviors</th>
<th>Effect</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized behavior ( ( S'_1 \subseteq S_1 ) and ( S'^* \subseteq S^* ) )</td>
<td>addition of state variables which are used in existing behaviors (to refine them)</td>
<td>( b_{SPL} ): Book Copy ( \text{is available} ) ( \text{is unavailable} ) &lt;br&gt; Book Copy ( \text{is available} ) ( \text{is unavailable} ) &lt;br&gt;Borrowable? ( \text{yes, no} )</td>
</tr>
<tr>
<td>Template-instantiated behavior ( ( S_1 \rightarrow S'_1 ) and ( S^* \rightarrow S'^* ) )</td>
<td>different state variables and their values which change similarly</td>
<td>( b_{SPL} ): Item ( \text{is available} ) ( \text{is unavailable} ) &lt;br&gt; Item ( \text{is available} ) ( \text{is unavailable} ) &lt;br&gt; Disk Location ( \text{on shelf} ) ( \text{off shelf} )</td>
</tr>
</tbody>
</table>

A _specialized behavior_ \( b_P = ( S'_1, < e >, S'^* ) \) refines the initial state and/or the final state of the behavior by looking at additional state variables or more detailed values\(^2\).

\(^2\) We refer to events as the triggers of transformations. Thus, we currently do not involve them in the definitions of specialized and template-instantiated behaviors.
Formally stated, $S'_1 \subseteq S_1$ and $S'^* \subseteq S^*$. This means that the set $S'_1$ is a refinement (specialization) of the set $S_1$ and the same for $S'^*$ and $S^*$.

A **template-instantiated behavior** $b_{\phi}=(S'_1, \ <e'>, S'^*)$ works on different state variables than the original behavior ($b_{\text{SPL}}$), but its effect in terms of transformation is similar. Formally stated, there are mappings $m_i:S_i \rightarrow S'_i$, $m^*:S^* \rightarrow S'^*$, such that $m_i$ and $m^*$ are bijections (total, onto, and 1-to-1). We denote these mappings by $S_i \mapsto S'_i$ and $S^* \mapsto S'^*$.

## 4 Identifying Variants through Variability Mechanisms

The above framework describes the relations between core assets and product artifacts in the element dimension, in terms of state variables and values. Due to the great popularity of object-orientation in the software engineering community, we adapt the general framework described in Section 3 to object-oriented artifacts (design or code). The mapping between the object-oriented terminology and the ontological concepts is straightforward: (objects of) classes can be mapped to things, attributes – to state variables, types – to (potential) values, and operations (methods) – to behaviors.

Below we adapt the notions of specialized and template-instantiated behaviors to object-oriented concepts in order to define specialization and/or template instantiation. A method to use these adaptations in practice and preliminary results are described in Section 5.

### 4.1 Basic Definitions and Notations

We consider a set of classes, along with their attributes and operations. As a simple example for demonstrating our basic definitions, consider a class $BookCopy$. Its attributes are:

- **BorrowingPeriod** (specifying for how many days the book copy can be borrowed);
- **AvailabilityStatus** (specifying whether the book is available or not, i.e., borrowed);
- **ReturnDate** (specifying the date in which the book copy is expected to be returned, if it is borrowed).

The operations of the class include:

- The constructor (which mainly initializes the `BorrowingPeriod`);
- A borrow operation (changing the `AvailabilityStatus` from true to false and the `ReturnDate` to the current date plus `BorrowingPeriod`);
- A return operation (changing the `AvailabilityStatus` from false to true and resetting the `ReturnDate`).

We use the following representation of classes.

---

3 Besides those concepts, object-orientation follows some fundamental principles (e.g., SOLID) and processes (e.g., refactoring). However, since we do not explicitly deal with writing code, but with identifying places in code in which application of variability mechanisms can increase reuse, those principles and processes are out of scope of our work.
**Definition 1.** An **attribute** att is represented by a pair (name, vals), where name is the attribute name and vals is its type, representing the values it can assume.

We denote by att.name and att.vals the first and second constituents of att, respectively. In the BookCopy example, we have the following three attributes:

\[ a_1 = (\text{BorrowingPeriod}, \{i \mid i \text{ is of type int}\}); a_2 = (\text{AvailabilityStatus}, \{\text{true}, \text{false}\}); a_3 = (\text{ReturnDate}, \{d \mid d \text{ is of type Date}\}). \]

An operation o is represented by two descriptors. One, named shallow (external), is equivalent to the operation signature (interface) and includes what other classes or operations “know” regarding the behavior. The second, named deep, reflects the (internal) impact of the operation on attributes. In other words, it specifies the transformation performed by the operation. As shown later, the shallow and deep behavior descriptors enable specifying the behavior captured by the operation in terms of initial state (\(S_1\)), external events (\(<e>\)), and final state (\(S^*\)).

**Definition 2.** The **shallow behavior descriptor** of an operation o is denoted by \(b_{\text{shallow}}(o) = (\text{op_name}, \text{params}, \text{ret_type})\) where op_name is the operation name, params is a set of pairs (name, vals) denoting the operation’s parameters and types, and ret_type is the returned type (all types are in the form of sets of possible values).

We denote by \(b_{\text{shallow}}(o).\text{op_name}, b_{\text{shallow}}(o).\text{params}, \) and \(b_{\text{shallow}}(o).\text{ret_type}\) the three constituents of \(b_{\text{shallow}}(o)\), respectively. For example, the shallow behavior descriptors of the three operations of class BookCopy are:

\[
\begin{align*}
 b_{\text{shallow}}(o_1) & = (\text{BookCopy}, \{(\text{newBorrowingPeriod}, \{i \mid i \text{ is of type int}\})\}, \emptyset); \\
 b_{\text{shallow}}(o_2) & = (\text{Borrow}, \emptyset, \emptyset); \\
 b_{\text{shallow}}(o_3) & = (\text{Return}, \emptyset, \emptyset).
\end{align*}
\]

**Definition 3.** The **deep behavior descriptor** of an operation o is denoted by \(b_{\text{deep}}(o) = (\text{att_used}, \text{att_modified})\) where att_used is the attributes involved in the operation along with their possible values (in the operation) before being changed, and att_modified is the set of the attributes being modified by the operation along with the values assigned to them.

We denote by \(b_{\text{deep}}(o).\text{att_used} \) and \(b_{\text{deep}}(o).\text{att_modified}\) the two constituents of \(b_{\text{deep}}(o)\), respectively. The deep behavior descriptors of the class BookCopy are:

\[
\begin{align*}
 b_{\text{deep}}(o_1) & = (\emptyset, \{(\text{BorrowingPeriod}, \text{newBorrowingPeriod}), (\text{AvailabilityStatus}, \text{true}), (\text{ReturnDate}, \text{null})\}); \\
 b_{\text{deep}}(o_2) & = (\{(\text{AvailabilityStatus}, \text{true})\}, \{(\text{AvailabilityStatus}, \text{false}), (\text{ReturnDate}, \text{now}() + \text{BorrowingPeriod})\}); \\
 b_{\text{deep}}(o_3) & = (\{(\text{AvailabilityStatus}, \text{false})\}, \{(\text{AvailabilityStatus}, \text{true}), (\text{ReturnDate}, \text{null})\})
\end{align*}
\]

Attributes and operations are used to represent the **ontological counterparts**, which are state variables and behaviors of things. State variables have names and a range of associated values, as reflected by the name and range of values (type) of attributes.

Ontological behaviors have the form \(b = (S_1, <e>, S^*)\), reflected by \(S_1 = \text{att_used} \cup \text{params}\) (namely, all attributes and parameters being used by the operation before being modified by it) and \(S^* = \text{att_modified} \cup \text{ret_type}\) (i.e., all attributes modified by...)

\[^4\text{Note that since we are interested in an external view of a behavior, as reflected in the triplet } (S_1, <e>, S^*), \text{ we ignore the impact of the behavior on the local variables of the operation.}\]
the operation and the returned parameter). <e> is derived from the semantics of op_name.

4.2 Similarity-Based Relations

In order to identify whether operations, and consequently classes, can be considered variants of each other or variants of core asset classes, we use the notion of similarity. Similarity between classes is based on the similarity of their attributes and operations. Similarity of attributes is calculated with their names and potential values. Similarity of operations is calculated with respect to shallow and deep behavior descriptors. We therefore assume the existence of a similarity measure of the following form:

Definition 4. Let C be a set of classes \{C_1, …, C_n\}. A similarity measure for C is a function from pairs of class constituents (attributes, shallow behavior descriptors, or deep behavior descriptors) to a Boolean value indicating whether the pair of constituents is similar or not. Formally expressed:

\[
\text{sim}: \text{Atts}(C) \times \text{Atts}(C) \cup \text{B}_{\text{shallow}}(C) \times \text{B}_{\text{shallow}}(C) \cup \text{B}_{\text{deep}}(C) \times \text{B}_{\text{deep}}(C) \rightarrow \{0, 1\}^5,
\]

where:

\[
\begin{align*}
\text{Atts}(C) & = \{(a.\text{att}_\text{name}, a.\text{vals}) \mid a \text{ is an attribute of } C_i \in C\} \\
\text{B}_{\text{shallow}}(C) & = \{b_{\text{shallow}}(o) \mid o \text{ is an operation of } C_i \in C\} \\
\text{B}_{\text{deep}}(C) & = \{b_{\text{deep}}(o) \mid o \text{ is an operation of } C_i \in C\}
\end{align*}
\]

Using the example of class BookCopy, assume another class MediaItem with an attribute Location of type enumeration and possible values: on-the-shelf and off-the-shelf (meaning borrowed). The operations of this class are similar to those of BookCopy, borrow and return, where borrow changes the Location from on-the-shelf to off-the-shelf, and return changes this attribute in the opposite direction. We can argue that the similarity of the attributes Location and AvailabilityStatus is 1 due to their isomorphic types and the similar roles they play in borrow and return. The similarity of the operations, in terms of both shallow and deep, is also 1.

Based on the similarity measure above, we define two relations between behaviors: inclusion similarity (\(\leq_{\text{sim}}\)) and replacement similarity (\(\rightarrow_{\text{sim}}\)). Using these relations we can adapt the notions of specialized and template-instantiated behaviors to object-oriented design and programming.

Definition 5 (inclusion similar): Operation \(o_i\) of class \(C_i\) is inclusion similar to operation \(o_j\) of class \(C_j\) if the components of \(b_{\text{shallow}}(o_i)\), \(\text{op-name, params, ret-type}\), are included in the components of \(b_{\text{shallow}}(o_j)\) and the components of \(b_{\text{deep}}(o_i)\), \(\text{att-used and att-modified}\), are included in the components of \(b_{\text{deep}}(o_j)\), up to the similarity measure for \(\{C_i, C_j\}\). We denote this by \(o_i \leq_{\text{sim}} o_j\).

Definition 6 (replacement similar): Operation \(o_i\) of class \(C_i\) is replacement similar to operation \(o_j\) of class \(C_j\) if there is a bijection \(b\), which maps each component \(c\) of

\[
\text{Note that for simplicity, we assume that the similarity measure returns a Boolean value, 0 (different) or 1 (similar), rather than a range of values indicating the degree of similarity. In the method implementation, we realized the similarity measure by ranges and thresholds.}

6 By ‘up to the similarity measure’ we mean that for each pair of components of either shallow or deep behavior descriptors, A and B, B includes A means that for each element \(a \in A\) there is an element \(b \in B\), such that \(\text{sim}(a, b) = 1\).
b_{shallow}(o_i) and b_{deep}(o_i), (namely, op-name, params, ret-type, att-used and att-modified) to a corresponding component b(c) of b_{shallow}(o_j) and b_{deep}(o_j), respectively, such that \( \text{sim}(c, b(c)) = 1 \). We denote this by \( o_i \sim_{\text{sim}} o_j \).

Using the examples of classes BookCopy and MediaItem, assume an extra Boolean attribute of MediaItem, borrowable?, indicating whether the media item is borrowable or not. The operation borrow changes the media item’s location from “on-the-shelf” to “off-the-shelf” if borrowable is true. It is easy to check that BookCopy.borrow \( \subseteq_{\text{sim}} \text{MediaItem}.borrow \), as op-name, params, ret-type, att-used and att-modified of BookCopy.borrow are similar (as explained above) to the corresponding components of MediaItem.borrow. The latter also has an additional attribute (borrowable?) which does not match any attributes of BookCopy.borrow. Thus, actually, BookCopy.borrow \( \subseteq_{\text{sim}} \text{MediaItem}.borrow \).

Now consider two classes: CatalogOfBooks and CatalogOfItems. Both have similar operations, such as add to the catalog and browse. However, they work on different objects: CatalogOfBooks works on objects of class BookCopy, while CatalogOfItems works on items (such as BookCopy and MediaItem). Defining a bijection mapping between Item and BookCopy, it can be shown that:

- \( \text{CatalogOfItems}.add \Rightarrow_{\text{sim}} \text{CatalogOfBooks}.add \)
- \( \text{CatalogOfItems}.browse \Rightarrow_{\text{sim}} \text{CatalogOfBooks}.browse \)

### 4.3 Identifying Specialization and Template Instantiation

So far we have discussed specialization and template instantiation on the behavior (or operation) level. However, these concepts are usually used on the class-level.

**Specialization** deals with refinement of behavior by addition of state variables (attributes) or by enhancing values of existing state variables. We concentrate here on cases where specialized classes do not violate the intended behaviors of the generalized classes (as may occur in code inheritance, e.g., through overriding). Since our approach is behavioral, we refer to attributes implicitly, through the changes they undergo (described in att-used and att-modified of the deep behavior descriptor). We next define specialization in terms of behaviors and inclusion similarity.

**Definition 7.** Let \( C_1, C_2 \) be classes. \( C_1 \) is a specialization of \( C_2 \) if for each operation \( o_2 \) of \( C_2 \) there is an operation \( o_1 \) of \( C_1 \) such that \( o_2 \) is inclusion similar to \( o_1 \) (\( o_2 \in_{\text{sim}} o_1 \)).

Note that the behavior of \( C_2 \) can be extended in \( C_1 \), namely, \( C_1 \) has operations with no counterparts in \( C_2 \), and operations from \( C_2 \) can be used as they are (without specialization) in \( C_1 \). As an example consider a variation of the class BookCopy, named BookCopy1, with an additional attribute – last – which indicates whether the book copy is considered the last to be borrowed. Borrowing the last book copy requires special handling, e.g., notifying the librarian. According to the definition above, BookCopy1 is a specialization of BookCopy.

As noted, template instantiation deals with type adaptation and generic behaviors and can be characterized using replacement similarity as follows:

**Definition 8.** Let \( C_1, C_2 \) be classes. \( C_1 \) is template instantiation of \( C_2 \) if:

- For each operation \( o_2 \) of \( C_2 \) there is an operation \( o_1 \) of \( C_1 \) such that \( o_2 \) is replacement similar to \( o_1 \) (\( o_2 \Rightarrow_{\text{sim}} o_1 \)).
For each operation $o_1$ of $C_1$ there is an operation $o_2$ of $C_2$ such that $o_1$ is replacement similar to $o_2$ ($o_2 \sim_{\text{rep}} o_1$).

The class CatalogOfBooks is a template instantiation of class CatalogOfItems.

5 Implementation and Preliminary Results

To evaluate our analysis, we developed and implemented a proof-of-concept, following the stages in Fig. 1. The inputs are object-oriented code artifacts belonging to different products ($P_1, \ldots, P_n$). The outputs are variability models specifying similar classes and the variability mechanisms associated with them. Note that we do not assume the existence of core assets. Instead we use our formal foundations in order to set the ground for the definition of core assets. We next elaborate on each stage of the implemented method and on some preliminary results.

Extract information: The input software artifacts (object-oriented code) are transformed (reverse engineered) into: (i) class diagrams (in XMI format) from which the attributes and shallow behavior descriptors can be extracted and (ii) Program Dependence Graphs (PDG)\(^7\) \[18\] (in JSON format\(^8\)) from which the deep behavior descriptors are extracted. These particular code representations have the ability to represent structure and behavior (rather than specific scenarios). Moreover, they are common and mature as can be reflected by the availability of a variety of tools to reverse engineer object-oriented code. Finally, these representations enabled ignoring low level details, such as comments and syntactic differences.

Fig. 2 and Fig. 3 depict the class diagram generated for our Book Copy class and the PDG of its return operation, respectively\(^9\). The class diagram further includes a class named Book, which describes the title of the book and its publication year. A Book may have many Book Copies.

![Fig. 2. An example of a class diagram taken as an input to our method](image)

While the attributes and the shallow behavior descriptors are directly and simply derived from the class diagrams, the extraction of deep behavior descriptors deserves special attention. We utilize only the vertices getField and putField from the PDGs (examples are circled in Fig. 3). These vertices get or set a value of static/object field. The use and modification of attributes may be done directly in the operation or

---

\(^7\) PDG explicitly represents the data and control dependencies of a program.

\(^8\) JSON – JavaScript Object Notation – is a lightweight data-interchange format.

\(^9\) These diagrams were reverse engineered using Visual Paradigm (http://www.visual-paradigm.com/) and JavaPDG (http://selserver.case.edu:8080/javapdg/), respectively.
through other operations invoked by the operation at hand. Hence, we consider a third type of vertices – CallSite (not included in the figure). Through these vertices our method can recursively check getField and putField vertices in the invoked operations. Listing 1 describes how the initial state of a (behavior induced by a) given operation o is derived. The final state is similarly computed on put field vertices.

```
OperationInitialState(operation o, Set s1): Set
    Set combinedS1=s1 ∪ retrieveGetField(o)
    Set invoked=retrieveCallSite()
    For each op in invoked
        combinedS1=OperationInitialState(op, combinedS1)
    return combinedS1

Listing 1. Deriving the initial state of a given operation
```

![Fig. 3. An example of a PDG for the return operation of BookCopy](image)

**Analyze Commonality/Variability:** Similarity plays an important role to analyze variability. We decided to measure the similarity of names (attributes, operations, and parameters) by using semantic metrics.\(^{10}\) Type similarity was equality, namely,

\(^{10}\) We assume that attributes, operations, and parameters have meaningful names (potentially including several words separated by underscores or capital letters). Otherwise, preprocessing is needed before our method can be applied.
two types are similar if and only if they are the same. Attribute similarity was computed as the weighted average of name and type similarities.

Shallow similarity was computed as the weighted average of the similarities of their operation names, parameters, and returned types. The weights can be manually tuned by software designers to reflect the specific characteristics of the given products.

Deep similarity was calculated based on the symmetric difference (for multisets) of the attributes used and attributes modified. Formally expressed, let \( \text{AU}_o \) and \( \text{AM}_o \) be the attributes_used and attributes_modified of operation \( o \), respectively, and \( \text{AU}_{o'} \), \( \text{AM}_{o'} \) – the attributes_used and attributes_modified of operation \( o' \), respectively. The deep similarity of \( o \) and \( o' \) is defined as:

\[
1 - \frac{|\Delta (\text{AU}_o, \text{AU}_{o'})| + |\Delta (\text{AM}_o, \text{AM}_{o'})|}{|\text{AU}_o| + |\text{AU}_{o'}| + |\text{AM}_o| + |\text{AM}_{o'}|},
\]

where \( \Delta \) is the symmetric difference operator for multisets, calculated as \((\text{AU}_o - \text{AU}_{o'}) \cup (\text{AU}_{o'} - \text{AU}_o)\) and \( |A| \) is the number of elements in \( A \).

For retrieving similar attributes, shallow behavior descriptors, and deep behavior descriptors, we used a hierarchical agglomerative clustering algorithm [17]. This algorithm starts by putting each element in a separate cluster and merges in each iteration the closest clusters, namely, clusters whose average similarity is the highest. Those clusters were used for examining inclusion similarity and replacement similarity. Note that the method associates a variability mechanism (specialization or template instantiation) if potentially the classes satisfy the mechanism relation (inclusion or replacement similarity, respectively).

**Model Variability:** To visualize the analysis results, we use Orthogonal Variability Modeling (OVM) [24]. OVM represents variability through the concepts of variation point and variant. A variation point, denoted by triangles, represents a variable item or a property of an item. A variant, denoted by a rectangle, defines a possibility to realize the variable item or property. OVM further supports relating variability information to software artifacts (such as requirements, design, and code) that are affected by the variability. For each cluster of similar classes, our method defines a variation point and associates with it the variability mechanisms utilized to identify the variants. The method further defines variants – one for each class in the cluster – and associates them with the variation point via an OR relation (see Fig. 4 for an example). Currently, we do not handle dependencies between variation points and variants.

---

**Legend:**
- **VP** [mechanism] Optional variation point
- **V** [class name] Variant
- **mechanism** Choice (between \( m \) to \( n \))

---

**Fig. 4.** Examples of the method outputs
Preliminary Evaluation. We explored the code of two open-source versions of a SuperSnake game\textsuperscript{11}: Supersnake1.0 (released in 2008, containing 8 classes) and Supersnake 2.0 (released in 2010, containing 14 classes, some of which are exact copies or clones of classes from version 1.0). Running the method on the representations reversed engineered from this code resulted in 20 cases of specialization and 18 cases of template instantiation\textsuperscript{12}. While 7 of the specialization and 5 of the template instantiation cases were trivial (i.e., included classes copied or cloned from the previous version), the rest of the cases were found relevant based on manual examination. Particularly, these cases pointed on high degrees of similarity between classes, in terms of their exhibited behaviors. We further observed some limitations which we plan to address in future research. First, the use of a general purpose vocabulary for measuring the semantic similarities led to irrelevant cases, such as menu (in the context of GUI) and food. These can be easily eliminated by using programming-related or domain-specific vocabularies. Second, the implemented method tries to locate for each element in one class similar elements in the other class without controlling multiplicities, thus leading to over generalization (e.g., classes with similar GUI controls). Finally, the presence of numerous getter and setter operations led to identification of template instantiation, regardless of the semantics of the corresponding transformations. This can be addressed by a more in-depth analysis of code representations and refinement of the similarity measure for deep behaviors.

6 Conclusions and Future Directions

Identification of variants of software artifacts is important for improving software development and maintenance. While current approaches mainly aim to avoid variants (through refactoring) or track them, we propose a method for utilizing variability mechanisms, and specifically specialization and template instantiation, to analyze and represent variability. The method is based on ontological foundations, which allow focusing on behaviors rather than on implementation. The method can be used to identify situations and places in software artifacts where different variability mechanisms may need to be applied in order to increase and systematize reuse. A proof-of-concept implementation of the method was applied to open-source code.

Immediate directions for future research are evaluating the method with software designers and developers and extending it to other well-known variability mechanisms, such as parameterization, configuration, and analogy. Another direction is refining the analysis of similarity of attributes and operations, e.g., by considering isomorphic types and semantics of deep descriptors. Furthermore, employing other representations in addition to class diagrams and PDGs may lead to a more wide-spread analysis. These directions will facilitate the development of a tool for an automatic construction of core assets for the analyzed software artifacts, based on the relations at the basis of different variability mechanisms.

\textsuperscript{11} The two versions were taken from http://sourceforge.net/

\textsuperscript{12} See details in http://mis.hevra.haifa.ac.il/~iris/research/VarMech/InhTmpAnalysis.xlsx.
References


