Comprehensive Variability Analysis of Requirements and Testing Artifacts

Michal Steinberger and Iris Reinhartz-Berger(✉)

Department of Information Systems, University of Haifa, Haifa, Israel
mnachm04@campus.haifa.ac.il, iris@is.haifa.ac.il

Abstract. Analyzing variability of software artifacts is important for increasing reuse and improving development of similar software products, as is the case in the area of Software Product Line Engineering (SPLE). Current approaches suggest analyzing the variability of certain types of artifacts, most notably requirements. However, as the specification of requirements may be incomplete or generalized, capturing the differences between the intended software behaviors may be limited, neglecting essential parts, such as behavior preconditions. Thus, we suggest in this paper utilizing testing artifacts in order to comprehensively analyze the variability of the corresponding requirements. The suggested approach, named SOVA R-TC, which is based on Bunge's ontological model, uses the information stored and managed in Application Lifecycle Management (ALM) environments. It extracts the behavior transformations from the requirements and the test cases and presents them in the form of initial states (preconditions) and final states (post-conditions or expected results). It further compares the behavior transformations of different software products and proposes how to analyze their variability based on cross-phase artifacts.

Keywords: Variability analysis · Ontology · Software reuse · Software product lines · Application lifecycle management

1 Introduction

Variability analysis deals with determining the degree of similarity of different software artifacts, commonly in order to improve the effectiveness and efficiency of their development and maintenance through increase of reuse [7]. Variability analysis is extensively studied in the field of Software Product Line Engineering (SPLE) [11, 21], where *variability* is considered "an assumption about how members of a family may differ from each other" [28]. Variability analysis is known as time consuming and error-prone. Thus, various studies have suggested automatizing variability analysis using different software development artifacts. Many of these studies concentrate on analyzing the differences of requirements (e.g., [2, 12, 15]), perceiving reuse of requirements very important since requirements are essential in all development approaches and elicited and specified early in the software development lifecycle [11]. These studies frequently apply semantic, syntactic, metric-based, or graph-based similarity measurements and utilize clustering algorithms. The result is presented in a form of variability models, most notably feature diagrams [14]. Other types of software artifacts are also utilized in order to analyze variability, e.g., architecture [1] and code [3]. In contrast, testing artifacts seem to attract less attention in variability analysis [10]. This may be due to their reliance on other development artifacts (i.e., the requirements that they aim to test) or their description of specific scenarios (that sometimes include particular values or conditions). Moreover, to the best of our knowledge, utilizing different types of development artifacts in order to coherently analyze variability has not been studied. We claim that analyzing artifacts from different, but related, development phases may result in more comprehensive variability analysis outcomes that better represent the similarities and differences among software products and may consequentially increase reuse and improve software development and maintenance.

To this end, we propose in this paper to utilize information stored and managed on requirements and testing artifacts in existing software development tools. These kinds of artifacts are highly related in most development approaches and refer to software behaviors rather than to concrete implementations. Particularly, we explore Application Lifecycle Management (ALM) environments whose aim is to plan, govern, and coordinate the software lifecycle tasks. Although ALM environments are geared towards development of single products, we propose here to utilize them in order to analyze the variability of different software products managed in their repository. Particularly, we introduce a method, named Semantic and Ontological Variability Analysis based on Requirements and Test cases (or SOVA R-TC for short), that extracts software behaviors from requirements and testing artifacts, enables their comparison to other behaviors at different level of abstraction, and identifies variants of similar behaviors. Those variants set the ground for comprehensive variability analysis.

The rest of the paper is structured as follows: Sect. 2 reviews the background and related work, motivating the need to analyze variability of both requirements and testing artifacts. Section 3 elaborates on the suggested approach, while Sect. 4 presents insights from preliminary analysis of the approach outcomes. Finally, Sect. 5 concludes and provides directions for future research.

2 Background and Literature Review

2.1 Application Lifecycle Management (ALM)

Application Lifecycle Management (ALM) environments [15, 16] aim to support the development of software products from their initial planning through retirement. Their main advantages are: (1) maintaining high level of traceability between artifacts produced in different development phases, e.g., requirements and testing artifacts; (2) reporting on the development progress in real time to different stakeholders; and (3) improving stakeholders' communication across development tasks, e.g., developers can easily access the complete information about the failure of a test case and its results which led to finding defects (bugs). Due to those benefits, different frameworks and implementations of ALM can be found in the industry. Most of them support software requirements definition and management, software change and configuration management (SCM), software project planning, quality management (testing), and defect management. The first generation of environments, called ALM 1.0, integrates a few individual tools for helping stakeholders perform their tasks. The next generation, called ALM 2.0, proposes a holistic platform (rather than a collection of tools) for coordinating and managing development activities [15].

Although ALM environments typically relate to development of single products, BigLever [6] – a leading vendor of product line solutions – has already suggested *multi-phase* as one of three dimensions in their SPLE framework (the other two dimensions are *multi-baseline* referring to evolution of artifacts over time and *multi-product* referring to the diversity of products in the same software product line). The multi-phase dimension directly refers to the development lifecycle phases. It concerns consistency and traceability among asset variations in different lifecycle phases. Yet, this dimension addresses the development of single software products.

A recent industrial survey [5] 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 core assets are developed). In those scenarios the information stored in ALM environments for different software products can be utilized to analyze their variability.

Although no study suggests utilizing ALM environments for analyzing variability, different methods have been suggested for analyzing variability of different types of software artifacts stored in ALM environments, most notably requirements [10]. Next we review relevant studies on variability analysis at different development phases, concentrating of requirements engineering and testing artifacts.

2.2 Variability Analysis at Different Development Phases

Recently, Bakar et al. [4] conducted a systematic literature review on feature extraction from requirements expressed in a natural language. The main conclusions of this review is that most studies use Software Requirements Specifications (SRS) as inputs, but product descriptions, brochures, and user comments are also used due to practical reasons. The outputs of the suggested methods are commonly feature diagrams [14], clustered requirements, keywords or direct objects. Moreover, the extraction process can be divided into four phases: (1) requirements assessment, (2) terms extraction (using different techniques, such as algebraic models, similarity metrics, and natural language processing tools), (3) features identification, and (4) feature diagram (or variability model) formation. Most studies automatize the second phase of terms extraction, while the other phases are commonly done manually. A work that addresses the automatization of phases 3 and 4 in addition to that of phase 2 is SOVA (Semantic and Ontological Variability Analysis) which analyzes requirements variability based on ontological and semantic considerations [20, 22]. Due to the high relevance of SOVA to this work, we elaborate on it in Sect. 2.3.

In contrast to requirements, testing artifacts seem to attract less attention in variability management [10]. According to [10], only FAST – Family-Oriented Abstraction, Specification and Translation – can be considered covering the full lifecycle phases, from requirements engineering to testing. However, this approach concentrates on documentation and representation of variability and not on its analysis.

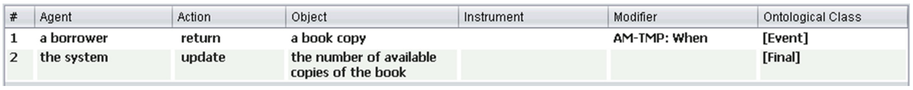
Other development artifacts, e.g., design artifacts [1, 17] and code [24], have also been analyzed to find differences between software products. A few approaches, e.g., [1, 25], further propose utilizing several distinct sources of information for analyzing variability. However, these sources commonly belong to the same lifecycle phase (e.g., textual feature descriptions and feature dependencies belonging to the requirements engineering phase; or software architecture and plugin dependencies belonging to the design phase). In our work, we aim to explore how analyzing the variability of artifacts from different lifecycle phases, particularly, requirements engineering and testing, can contribute to understanding the differences between various software products.

2.3 Semantic and Ontological Variability Analysis (SOVA)

SOVA aims to analyze variability among different software products based on their textual requirements. Feature identification relies on software behaviors extraction [20, 22]. For each behavior, the initial states (pre-conditions), the external events (triggers), and the final states (post-conditions) are identified. This is done by parsing the requirement text utilizing the Semantic Role Labeling (SRL) technique [13]. Six roles that have special importance to functionality are used in SOVA: (1) Agent – Who performs?; (2) Action – What is performed?; (3) Object – On what objects is it performed?; (4) Instrument – How is it performed?; (5) Temporal modifier – When is it performed?; And (6) Adverbial modifier – In what conditions is it preformed?

Based on these roles, the different phrases of the requirements (called vectors) are classified into initial states, external events, and final states. ***External events*** are: (1) Action vectors (i.e., vectors identified by verbs) whose *agents* are *external* and their *actions* are *active*, or (2) Action vectors whose *agents* are *internal* and their *actions* are *passive*. Oppositely, states are: (1) Action vectors whose *agents* are *internal* and *actions* are *active*, or (2) Action vectors whose *agents* are *external* and *actions* are *passive*. The decision whether a vector is classified as an initial state or a final state is done according to the place of the vector with respect to other vectors classified as external events. Particularly, ***initial states*** are: (1) Action vectors classified as states that appear *before* the *first* external events in the requirements, or (2) Non-action vectors (identified by temporal or adverbial modifiers) that appear *before* the *first* external events in the requirements. Conversely, ***final states*** are action vectors classified as states that appear *after* the *last* external event in the requirements.

As an example, consider Fig. 1 which presents SOVA's parsing outcome for the requirement: *When a borrower returns a book copy, the system updates the number of available copies of the book*. In this case, no initial state is extracted; the external event is returning a book copy by a borrower; and the final state is derived from updating the number of available copies of the book.



#	Agent	Action	Object	Instrument	Modifier	Ontological Class
1	a borrower	return	a book copy		AM-TMP: When	[Event]
2	the system	update	the number of available copies of the book			[Final]

Fig. 1. An example of SOVA's parsing outcome

After parsing the requirements and classifying their parts into initial states, external events, and final states, SOVA enables comparison of requirements, belonging to different software products, based on different perspectives. In [20], two perspectives are mainly discussed: structural – in which focus is put (through controlling weights) on differences in the initial and final states – and functional – which concentrates on differences in the external events. The final outcomes of SOVA are feature diagrams organized according to selected perspectives.

Despite the benefits of SOVA to analyze the commonality and variability of software behaviors [22], its success heavily depends on the level of details of the requirements and especially on their ability to express the initial state, external event, and final state. From practice, it is known that requirements are not always complete and particularly do not explicitly specify all pre-conditions and post-conditions. Hence, we aim to overcome this limitation by using the corresponding testing artifacts. To motivate the need, consider the following two requirements which may appear in different library management systems to describe book return functionality:

1. When a borrower returns a copy of a book that can be pre-ordered by other borrowers, the system sends a message to the borrower who is waiting for this book.
2. When a borrower returns a book copy, the system updates the number of available copies of the book.

The requirements are similar since they both handle returning a book copy. However, they differ in their pre- and post-conditions. While the post-conditions are specified in the requirements ("the system sends a message to the borrower who is waiting for this book" for the first requirement and "the system updates the number of available copies of the book" for the second requirement), the preconditions in this example are not explicitly specified. Specifically, in the first case, the precondition that there is a borrower waiting for the returned book is not mentioned. As a result, the similarity of these requirements will not reflect the difference in their preconditions and the variability analysis will be negatively affected. As we claim next, utilizing test artifacts associated with those requirements may improve variability analysis (increasing or decreasing the similarity of the corresponding requirements).

3 The Suggested Approach – SOVA R-TC

Our working hypothesis is that what commonly matters to stakeholders involved in different development lifecycle phases, including requirement engineers and testers, is the expected behavior of the implemented software. This premise is manifested by concepts such as functional requirements or functional testing. Therefore, the suggested approach, called SOVA R-TC, concentrates on functional requirements (hereafter requirements, for short) and the test cases associated to them. SOVA R-TC employs a view of a software product as a set of intended changes in a given application domain. We term such changes software behaviors. The approach uses an ontological model of behaviors based on concepts from Bunge's work [8, 9]. This model is described in Sect. 3.1. We further develop an ALM metamodel that concentrates on requirements and testing artifacts (Sect. 3.2) and use it for identifying variants based on the ontological model (Sect. 3.3).

3.1 The Ontological Model of Bunge

Bunge's work [8, 9] describes the world as made of things that possess *properties*. Properties are known via *attributes*, which are characteristics assigned to *things* by humans. Software products can be considered things.

To define or represent things and compare them, we have to define the point of view from which we wish to conduct the analysis. For example, libraries and car rental agencies may be perceived differently, since the first one deals with borrowing books that are different in terms of structure and functionality from cars, the focus of the second thing. However, these things can be perceived as very similar if we conduct the analysis from the point of view of checking out of items: both libraries and car rental agencies handle checking out of physical items (either books or cars).

To define the point of view, we use Bunge's notion of a *state variable* – a function which assigns a value to an attribute of a thing at a given time. The *state* of a thing is the vector of state variables' values at a particular point in time. The abstraction level of states can be controlled by selecting different sets of state variables, e.g., ISBN of a book and the license number of a car vs. an item identity. An *event* is a change of a state of a thing and can be external or internal: an *external event* is a change in the state of a thing as a result of an action of another thing (e.g., borrowing a book by a person or renting a car by a client), whereas an *internal event* arises due to an internal transformation in the thing (e.g., operations triggered by the library itself or the car rental agency). Correspondingly, a state can be stable or unstable: a *stable state* can be changed only by an external event while an *unstable state* may be changed by an internal event.

Bunge's ontological concepts have been widely adapted to conceptual modeling in the context of systems analysis and design [26, 27]. In [23], Bunge's ontological model is suggested to define software behavior as a triplet of an initial state describing the stable state the system is in before the behavior occurs, a sequence of external events that trigger the behavior, and a final state specifying the stable state the system reaches after the behavior terminates. Both initial and final states are described with respect to relevant state variables, allowing for variability analysis in different granularity levels. Formally expressed:

Definition 1 (Behavior). Given a stable state s_1 and a sequence of external events $<e_i>$, *a behavior* is a triplet (s_1, $<e_i>$, s^*), where s^* is the first stable state the thing reaches when it is in state s_1 and the sequence of external events $<e_i>$ occurs. s_1 is termed the *initial state* of the behavior and s^* – the *final state* of the behavior. s_1 and s^* are defined over a set of state variables $SV = \{x_1 \ldots x_n\}$, namely, s_1 and s^* are assignments to $x_1 \ldots x_n$.

3.2 An ALM Metamodel

In order to use Bunge's concepts as a basis for comprehensively analyzing variability of software products, we turn now to the introduction of a partial ALM metamodel that depicts the characteristics of requirements and testing artifacts, as well as their relations. Exploring IBM's Collaborative Lifecycle Management (CLM) solution[1], which is one of the leading ALM environments [29], and different ISO standards, most notably, ISO/IEC/IEEE 29119-3 on software testing documentation [19], we drafted the metamodel in Fig. 2. A requirement exhibits just a textual description (in a natural language), besides some identification. The testing artifacts are described via test cases. According to [19], a test case is a set of "preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information". A test case precondition describes "the required state of the test environment and any special constraints pertaining to the execution of the test case," whereas inputs are the "data information used to drive test execution." An expected result is "observable predicted behavior of the test item under specified conditions based on its specification or another source." Later in the standard, inputs are defined as actions "required to bring the test item into a state where the expected result can be compared to the actual results." This is in-line with the realization of inputs in existing ALM environments as fields named actions or test steps.

The mapping of the test case related elements to Bunge's terminology is quite straightforward: the preconditions define the initial state of the behavior (test case scenario), but also some technical constraints (e.g., regarding the environment); the inputs are the events that trigger ("drive") the behavior ("execution"); and the expected results partially[2] define the final states of the behavior.

Considering the two requirements mentioned earlier, Table 1 depicts a possible test case for each one of them. Generally, a single requirement may be validated by several test cases, e.g., a test case that validates the main scenario and test cases for different exceptions. Similarly, a single test case can validate more than one requirement, e.g., a test case that validates an exception common to different behaviors (requirements). Thus, the relations between requirements and test cases are many-to-many.

Given the suggested metamodel, a requirement or a test case is represented by the parts of the behavior they represent. For a requirement s_1, <e>, s* are extracted using SOVA, while for a test case s_1 is the set of preconditions, <e> is the sequence of inputs, and s* is the set of expected results.

Definition 2 (A Requirement). A *requirement*[3] R is represented by R = (rs_1, <re>, rs*), where rs_1 is the initial state of the behavior described in the requirement, <re> is the sequence of events triggering that behavior, and rs* is the final state of that behavior.

[1] http://www-03.ibm.com/software/products/en/ratlclm.

[2] The reason why the expected results only partially define the final state is that expected results concentrated on "observable predicted behavior of the test item," and not necessarily refer to internal changes.

[3] Remember we refer here only to functional requirements.

Table 1. Examples of test cases in the form of preconditions, inputs, and expected results

Requirement	Preconditions	Inputs	Expected results
When a borrower returns a copy of a book that can be pre-ordered by other borrowers, the system sends a message to the borrower who is waiting for this book	- The book can be pre-ordered by other borrowers - A borrower is waiting for the book to be returned	A borrower returns a book copy	The system sends a message to the borrower who is waiting for this book
When a borrower returns a book copy, the system updates the number of available copies of the book	——	A borrower returns a book copy	The system updates the number of available copies of the book

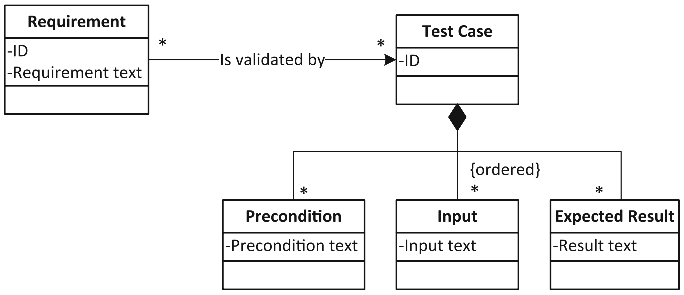


Fig. 2. A metamodel specifying the information on requirements and test cases

Definition 3 (A Test Case). A *test case* TC is represented by TC = (pre, <inp>, rst), where pre is the set of the preconditions of the behavior tested in the test case, <inp> is an ordered set of inputs, and rst is a set of expected results of the tested behavior.

In this paper, we assume that the test case inputs are actually realizations of the requirement external events (describing how the external events are captured by the system) and therefore we concentrate on the state of the system before the behavior occurs (as specified both in the initial states of the requirements and the preconditions of the test cases) and the state of the system after the behavior occurs (as specified both in the final states of the requirements and the expected results of the test cases). Note that these two parts of the behavior (s_1 and s*) represent states and hence can be perceived as defined by possible assignments to state variables. Returning to the examples in Table 1, the precondition of the first test case refers to two assignments of state variables – the book can be pre-ordered (book = can_be_preordered) and a borrower is waiting for the book (borrower = is_waiting_for_the_book). Each of the post-conditions of the two test cases refers to an assignment to a state variable: sending a message (message_to_the_borrower = sent) for the first test case and updating the number of available copies (number_of_available_copies = increased_by_1) for the second test case. Next we formally define states (e.g., rs_1, rs*, pre, and rst) as sets of pairs describing assignments to state variables.

Definition 4 (States). Let $SV = \{x_1, \ldots x_n\}$ be a set of state variables, such that $Dom(x_i) = \{v_{i1}, v_{i2}, \ldots\}$ is the possible values (domain) of x_i. A *state* s is defined as $s = \{(x_i, v_{ij}) | \; x_i \in SV \text{ and } v_{ij} \in Dom(x_i)\}$.

Since the suggested method assumes descriptions of behavior in a natural language (in the form of requirements and test cases), the extraction of state variables and assignments uses a natural language processing technique. Particularly, using the Semantic Role Labeling (SRL) technique mentioned in Sect. 2.3, the state variables are extracted from the object parts of the phrases, while the assignments are extracted from the action parts. The examples above follow these rules.

3.3 Behavior Transformations Deduced from Requirements and Test Cases

Using Bunge's ontological model and the metamodel introduced above, we aim to explore the relations between a requirement and its associated test cases in order to improve variability analysis. Particularly, SOVA R-TC compares the different parts of behaviors as specified in the requirements and their associated test cases.

While requirements describe what the system should do, namely, specify the high level functionality of the system, test cases detail how the functionality should be tested. Therefore, the initial state of a requirement may be different from the preconditions of the test cases associated to it. Particularly, the initial state of the requirement may be missing or the preconditions of the test cases may refine the initial state of the requirement. Similarly, the expected results of the test cases may differ (at least in the level of details) from the final state of the requirements. We do not refer to these differences as inconsistencies, but as differences in scope or in level of specification. Moreover, a single requirement may be associated to different test cases, each of which describes a different scenario to be tested. We thus consider the intersection of these scenarios (which describes the characteristics of the behavior rather than of a particular scenario/test case) and unify this intersection with the requirement in order to enrich the specification of the initial and final states. This is expressed through the notion of behavior transformation defined next.

Definition 5 (Behavior Transformation). Given a requirement $R = (s_1, \langle e \rangle, s^*)$ and a set of test cases $\{TC_1, \ldots TC_n\}$ associated to it, such that $TC_i = (\langle pre_i \rangle, \langle inp_i \rangle, \langle rst_i \rangle)$, we define the *behavior transformation* bt as (uis, ufs), where $uis = s_1 \cup_{i=1..n} pre_i$ is the unified initial state of the behavior and $ufs = s^* \cup_{i=1..n} rst_i$ is its unified final state.

For exemplifying behavior transformations, consider Table 2 which refers to a single requirement with two associated test cases. The initial state of behavior does not explicitly appear in the requirement and thus SOVA does not extract it. The final state generally refers to update of a certain variable (the number of available copies of the book). Using the two test cases associated to this requirement, we learn on two possible scenarios: one in which "no borrower pre-ordered the book" and the other in which "at least one borrower is waiting for the book." In both scenarios it is assumed that "the book was borrowed" and thus we can conclude that this assignment of the state variable (book_status) characterizes the behavior rather than provides technical conditions of specific test cases. As a result the unified initial state of the behavior transformation is "the book was borrowed." For similar reasons, the expected result of notifying the first borrower who is waiting for the book about the arrival is considered scenario-specific and hence the unified final state of the behavior transformation refers only to the state variable "number of available copies of the book." Here both requirement and test cases refer to this state variable, but slightly differently: the requirement generally refers to the need to update this state variable, while the test cases refer to how this state variable needs to be updated – increase by 1. Assuming that test cases are more detailed than requirements and refer to state variables in a higher level of details, we adopt the assignment proposed to a state variable by test cases. In other words, if the same state variable appears both in the requirements and in the test cases with different proposed assignments, the unified final state adopts the assignment suggested to the state variable in the test cases. Hence, in our case the unified final state of the behavior transformation is "the system increases the number of available copies of the book by 1."

Table 2. Examples of a requirement and two possible associated test cases

	s_1 (rs_1 or pre)	<e> (<re> or <inp>)	rs* or post (rs_1 or pre)
Requirement		a borrower returns a book copy	the system updates the number of available copies of the book
Test case 1	- The book was borrowed - No borrower pre-ordered the book	a borrower returns a book copy	- The system increases the number of available copies of the book by 1
Test case 2	- The book was borrowed - At least one borrower is waiting for the book	a borrower returns a book copy	- The system increases the number of available copies of the book by 1 - The first borrower who waits for the book is notified about the arrival
Behavior transformation	- The book was borrowed		- The system increases the number of available copies of the book by 1

3.4 Calculating the Similarity of Requirements Considering Their Associated Test Cases

The basis for analyzing variability in SOVA R-TC is identifying variants of similar behaviors. Thus, given a set of software products, each represented by requirements and their associated test cases, we calculate the similarity of behaviors as follows.

Definition 6 (Behavior Similarity). Given two requirements R_1 and R_2, their associated transformation behaviors bt_1 = (uis_1, ufs_1) and bt_2 = (uis_2, ufs_2), respectively, and a semantic similarity sim, the *behavior similarity,* $Sim_{R\text{-}TC}$, is the weighted average of the pair-wise semantic similarities of their unified initial and final states. Formally expressed:

$$\begin{aligned} & Sim_{R-TC}(R_1, R_2) \\ & = w_{uis} \cdot \frac{\sum_{x_i \in uis_1} max_{x_j uis_2} sim\left(x_i, x_j\right) + \sum_{x_i \in uis_2} max_{x_j uis_1} sim\left(x_i, x_j\right)}{|uis_1| + |uis_2|} \\ & + w_{ufs} \cdot \frac{\sum_{x_i \in ufs_1} max_{x_j ufs_2} sim\left(x_i, x_j\right) + \sum_{x_i \in ufs_2} max_{x_j ufs_1} sim\left(x_i, x_j\right)}{|ufs_1| + |ufs_2|} \end{aligned}$$

Where:

- w_{uis} and w_{ufs} are the weights of the unified initial and final states, respectively; $w_{uis} + w_{ufs} = 1$.
- $sim\left(x_i, x_j\right)$ is the semantic similarity of $x_i \, and \, x_j$ – assignments to state variables in the unified initial or final states of a requirement.
- $|uis_1|, |uis_2|, |ufs_1|, |ufs_2|$ are the numbers of assignments in the unified initial states of the two requirements and in the unified final states of the two requirements, respectively.

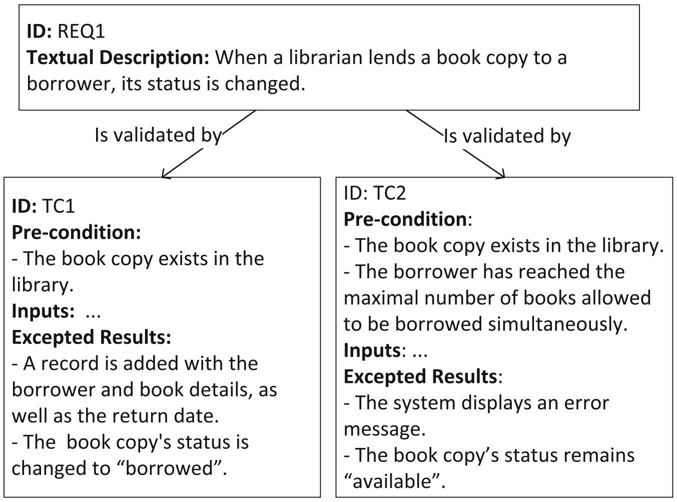
Simplifying this definition, behavior similarity is calculated by matching the most similar assignments both in the unified initial and final states of the compared behaviors. To this end, different semantic similarity measures can be used. Those measures are commonly classified as corpus-based or knowledge-based [18]. Corpus-based measures identify the degree of similarity based on information derived from large corpora, while knowledge-based measures use information drawn from semantic networks. Combining corpus- and knowledge-based semantic approaches, the measure suggested by Mihalcea et al. (MSC) [18] calculates sentence similarity by finding the most similar words in the same part of speech class. The derived word similarity scores are weighted with the inverse document frequency scores that belong to the corresponding word.

We made the calculation of behavior similarity flexible by introducing weights of the unified initial and final states. Basically, we could assume $w_{uis} = w_{ufs} = 0.5$, however, we wanted to enable the analysts to fine tune the calculation, based on observations they may have, e.g., regarding the accuracy and completeness of the unified initial states vs. the unified final states of the compared software products.

As an example to the benefits of calculating requirements taking into consideration their associated test cases, consider Fig. 3. The two requirements, taken from different products, are similar in the sense that they handle borrowing books. However, the similarity value of SOVA is 0.5 and the similarity value of MCS is 0.6, pointing on medium similarity. Calculating the behavior similarity of these requirements in SOVA R-TC, using the same basic semantic metric (MCS), we get a higher value of 0.7 which better depicts the expected conclusion that the behaviors can be considered variants of each other.

4 Implications and Discussion

We developed a tool supporting SOVA R-TC and analyzed a set of requirements and their associated test cases. Based on these examples we extracted patterns that worth further discussion and explorations. Note that although we are interested in the differences between analyzing the variability based only on requirements (the output of SOVA) and additionally utilizing test cases (the output of SOVA R-TC), the patterns are presented and explained based on the relations between the requirements and the test cases in the compared behaviors. Particularly, we refer to the requirement using the


ID: REQ1
Textual Description: When a librarian lends a book copy to a borrower, its status is changed.

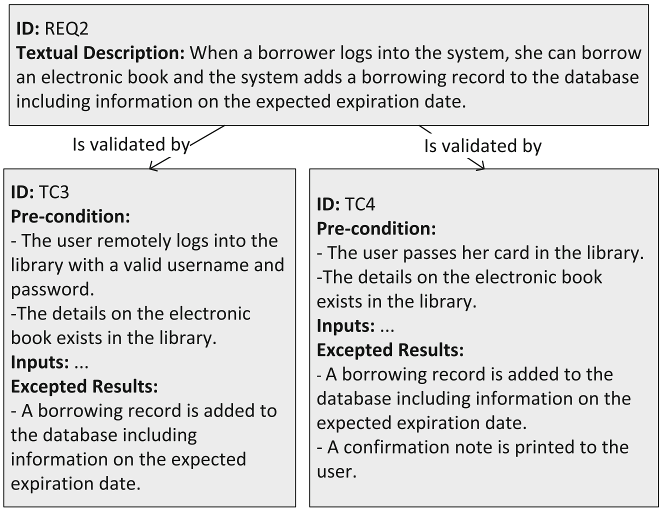
Is validated by

Is validated by

ID: TC1
Pre-condition:
- The book copy exists in the library.
Inputs: ...
Excepted Results:
- A record is added with the borrower and book details, as well as the return date.
- The book copy's status is changed to “borrowed”.

ID: TC2
Pre-condition:
- The book copy exists in the library.
- The borrower has reached the maximal number of books allowed to be borrowed simultaneously.
Inputs: ...
Excepted Results:
- The system displays an error message.
- The book copy’s status remains “available”.

(a) Product 1


ID: REQ2
Textual Description: When a borrower logs into the system, she can borrow an electronic book and the system adds a borrowing record to the database including information on the expected expiration date.

Is validated by

Is validated by

ID: TC3
Pre-condition:
- The user remotely logs into the library with a valid username and password.
-The details on the electronic book exists in the library.
Inputs: ...
Excepted Results:
- A borrowing record is added to the database including information on the expected expiration date.

ID: TC4
Pre-condition:
- The user passes her card in the library.
-The details on the electronic book exists in the library.
Inputs: ...
Excepted Results:
- A borrowing record is added to the database including information on the expected expiration date.
- A confirmation note is printed to the user.

(b) Product 2

Fig. 3. An example of requirements and their associated test cases from different products

notation of rs_1 and rs^*, and to the intersection of test cases via the notation of tcs_1 = and $tcs^* = \cap_{i=1..n} rst_i$. Remember that $s_1 = rs_1 \cup tcs_1$ and $s^* = rs^* \cup tcs^*$. Analyzing the four intersection possibilities between sets, Table 3 summarizes eight patterns, their characteristics, and implications on requirements similarity (and consequently on their variability analysis). Those patterns can be used in the future to study the impact of utilizing different types of software artifacts on the comprehensiveness of the variability analysis of their corresponding software products.

5 Summary and Future Research

Perceiving requirements as essential artifacts in software development, we advocate for utilizing test cases in order to better understand the similarity, and consequently the variability, of software products. Analyzing the similarity of requirements not just from their textual descriptions but also from their associated artifacts (test cases in our research), may improve understanding the requirements context and improve their reuse. This is especially important when the requirements are known to be incomplete

Table 3. Patterns of similarity in requirements

#	Name	Characteristics	Implications on similarity[a]	Explanations
1	Initial state refinement	$rs_1 \subseteq tcs_1$	↑↓	The requirements may become more or less similar, depending on the similarity of the added information ($tcs_1 – rs_1$ or $tcs^* – rs^*$)
2	Final state refinement	$rs^* \subseteq tcs^*$	↑↓	
3	Initial state consolidation	$tcs_1 \subseteq rs_1$	—	The similarity of the requirements are unchanged as $s_1 = rs_1$ or $s^* = rs^*$
4	Final state consolidation	$tcs^* \subseteq rs^*$	—	
5	Initial state exceptions	$rs_1 – tcs_1 \neq \emptyset \wedge tcs_1 – rs_1 \neq \emptyset$	↑↓	This can happen due to specification of exceptions in the test cases. The core characteristics of the requirements appear also in the test cases. The exceptions may increase or decrease requirements similarity
6	Final state exceptions	$rs^* – tcs^* \neq \emptyset \wedge tcs^* – rs^* \neq \emptyset$	↑↓	
7	Initial state conflict	$tcs_1 \cap rs_1 = \emptyset$	unexpectedly	This can happen due to reuse of generic test cases that are not directly related to the examined requirements. As a result the requirements similarity can unexpectedly be changed (increased or decreased)
8	Final state conflict	$tcs^* \cap rs^* = \emptyset$	unexpectedly	

[a]↑↓ = similarity changes (increases or decreases), — similarity unchanged.

or less detailed (e.g., in agile development approaches). The relevant information to do that is already stored and managed in Application Lifecycle Management (ALM) environments that are commonly used in software development. We suggest SOVA R-TC which utilizes ALM environments in order to extract the unified initial and final states of the behavior transformations and calculate the similarity of requirements considering the associated test cases. Based on this analysis, decisions on SPLE adoption (in extractive and reactive scenarios) can evidentially be taken.

In the future, we plan to further explore the patterns and evaluate their existence and potential meanings in different case studies. The evaluation will be done in comparison to existing methods and interviewing developers. We also intend to explore combination of patterns and examine similarity and variability of events (inputs) and how they impact requirements reuse, potentially adding patterns. Moreover, we plan to explore the impact of different types of relations between requirements and test cases (e.g., main scenarios vs. exceptions) on requirements similarity and variability.

References

1. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Extraction and evolution of architectural variability models in plugin-based systems. Soft. Syst. Model. (SoSyM) **13** (4), 1367–1394 (2013)
2. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems, pp. 45–54 (2012)
3. Alves, V., Matos Jr., P., Cole, L., Borba, P., Ramalho, G.L.: Extracting and evolving mobile games product lines. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 70–81. Springer, Heidelberg (2005)
4. Bakar, N.H., Kasirun, Z.M., Salleh, N.: Feature extraction approaches from natural language requirements for reuse in software product lines: a systematic literature review. J. Syst. Soft. **106**, 132–149 (2015)
5. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modeling of Software-Intensive Systems, pp. 7:1–7:8. ACM (2013)
6. BigLever: The systems and software product line engineering lifecycle framework (2013). http://www.biglever.com/extras/PLE_LifecycleFramework.pdf
7. Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media, Heidelberg (2005). Pohl, K. (ed.)
8. Bunge, M.: Treatise on Basic Philosophy, Ontology I: The Furniture of the World, vol. 3. Reidel, Boston (1977)
9. Bunge, M.: Treatise on Basic Philosophy, Ontology II: A World of Systems, vol. 4. Reidel, Boston (1979)
10. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009), pp. 81–90 (2009)
11. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley, Boston (2001)

12. Davril, J.M., Delfosse, E., Hariri, N., Acher, M., Cleland-Huang, J., Heymans, P.: Feature model extraction from large collections of informal product descriptions. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, pp. 290–300 (2013)
13. Gildea, D., Jurafsky, D.: Automatic labeling of semantic roles. Comput. Linguist. **28**(3), 245–288 (2002)
14. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report (1990)
15. Kim, J.A., Choi, S.Y., Hwang, S.M.: Process & evidence enable to automate ALM (Application Lifecycle Management). In: 9th IEEE International Symposium on Parallel and Distributed Processing with Applications Workshops (ISPAW 2011), pp. 348–351 (2011)
16. Lacheiner, H., Ramler, R.: Application lifecycle management as infrastructure for software process improvement and evolution: experience and insights from industry. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011), pp. 286–293 (2011)
17. Losavio, F., Ordaz, O., Levy, N., Baiotto, A.: Graph modelling of a refactoring process for product line architecture design. In: Computing Conference (CLEI) XXXIX Latin American, pp. 1–12 (2013)
18. Mihalcea, R., Corley, C., Strapparava, C.: Corpus-based and knowledge-based measures of text semantic similarity. In: The 21st National Conference on Artificial Intelligence (AAAI 2006), vol. 1, pp. 775–780 (2006)
19. ISO/IEC/IEEE 29119-3: Software and systems engineering—software testing—part 3: test documentation. International Organization for Standardization (2013)
20. Itzik, N., Reinhartz-Berger, I., Wand, Y.: Variability analysis of requirements: considering behavioral differences and reflecting stakeholders perspectives. IEEE Trans. Soft. Eng. (2016). doi:10.1109/TSE.2015.2512599
21. Pohl, K., Böckle, G., van der Linden, F.: Software Product-Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
22. Reinhartz-Berger, I., Itzik, N., Wand, Y.: Analyzing variability of software product lines using semantic and ontological considerations. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 150–164. Springer, Heidelberg (2014)
23. Reinhartz-Berger, I., Sturm, A., Wand, Y.: External variability of software: classification and ontological foundations. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 275–289. Springer, Heidelberg (2011)
24. Roy, C.K., Cordy, J.R.: A survey on software clone detection research (2007). http://maveric0.uwaterloo.ca/~migod/846/papers/roy-CloningSurveyTech Report.pdf
25. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 461–470 (2011)
26. Wand, Y., Weber, R.: On the deep structure of information systems. J. Inf. Syst. **5**(3), 203–223 (1995)
27. Wand, Y., Weber, R.: An ontological model of an information system. IEEE Trans. Soft. Eng. **16**, 1282–1292 (1990)
28. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, Boston (1999)
29. Wilson, N., Duggan, J., Murphy, T.E., Sobejana, M., Herschmann, J.: Magic quadrant for application development life cycle management. Gartner report (2015). http://www.gartner.com/technology/reprints.do?id=1-2A61Y68&ct=150218&st=sb