Published in Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing, Pau, France, 9-13 April 2018, which should be cited to refer to this work.

Program Understanding Using Ontologies and Dynamic Analysis

Javier Belmonte
eKlore Srl
Rue Voltaire 12
1201 Geneva, Switzerland
javier@eklore.ch

Philippe Dugerdil
Geneva School of Business Administration, HES-SO
Rue de la Tambourine 17
1227 Carouge, Switzerland
philippe.dugerdil@hesge.ch

ABSTRACT

No maintenance activity can be performed without understanding at least the part of the program that needs to be modified. Therefore, considering its cost, helping developers to understand programs is a must. Consequently, our research aims at building a business-related model of the program semantics, which is grounded in Perkinsfi research in psychology. After a short reminder of our model, whose performance in helping developers to understand programs has been presented elsewhere, this paper presents the automatic instantiation of the model. This rests on the ontology technology as well as on an innovative dynamic analysis technique. We present a use case to evaluate the performance of our technique.

CCS CONCEPTS

•**Software and its engineering** → **Software reverse engineering; Maintaining software;** •**General and reference** → *Experimentation;* •**Computing methodologies** → Ontology engineering;

KEYWORDS

Program understanding, reverse engineering, ontology, dynamic analysis

ACM Reference format:
Javier Belmonte and Philippe Dugerdil. 2018. Program Understanding Using Ontologies and Dynamic Analysis. In *Proceedings of ACM SAC Conference, Pau, France, April 9–13, 2018 (SAC'18),* 8 pages.
DOI: 10.1145/3167132.3167298

1 INTRODUCTION

Program understanding is central to the maintenance of software systems. Indeed, no maintenance activity can be performed without understanding at least the part of the program that needs to be modified [9]. Moreover, program understanding is a complex and costly task: it represents two-thirds of the maintenance expenses [18]. Consequently, we have focused our research on increasing the efficiency of the developers, particularly by helping them understand the program's source code during maintenance activities.

SAC'18, Pau, France

DOI: 10.1145/3167132.3167298

According to [23, 24], program understanding is built unconsciously and takes the form of some mental mapping between the elements of the problem domain (business domain or real world) and the elements of the system domain (source code). This mapping represents the understanding of the program being developed. Biggerstaff et al.'s definition of program understanding [8] still remains the reference definition used by the researchers in the field: "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."

Although popular, this definition is rather informal and can lead to many interpretations. In particular, it is not enough to guide us to elaborate a model to help the developers. This is why we complemented this definition with the results of Perkins' research in the psychology of human understanding [20]. The corresponding knowledge model, as well as its performance in helping developers to understand programs, has been presented elsewhere [5]. Because of this model's complexity, in this article, we focus on its automatic instantiation and on a case study to assess the performance of our technique.

1.1 Outline

In this paper, section 2 introduces our definition of program understanding. Section 3 is a reminder of the understanding artifact we proposed in [5] and presents the main ideas underlying the automatic production of the missing connections between the components of our artifact. Section 4 formalizes the ideas presented in section 3 and section 5 explains their implementation. Section 6 presents two experiments to evaluate this automated procedure. We present a survey of related works in section 7. A summary of the contributions made by our research and some future work ideas based on our experimental results are presented in section 8.

2 UNDERSTANDING

According to Perkins cited by Baron [3], understanding a situation involves three elements: a purpose(s), a structure, and the argumentation supporting the capacity of the structure to fulfill the purpose. Although the descriptions of the structure and the purpose of a situation depend on pre-existing knowledge, they do not depend on each other. But the argumentation does depend on the knowledge of the structure and the purpose. Indeed, the facts and beliefs that compose the argumentation connect the purpose to the components of the structure involved in its fulfillment. In summary, the structure and purpose are by themselves simply descriptions of

the situation to be understood, whereas the argumentation is what gives rise to understanding.

Program understanding in the context of maintenance activities means understanding a program's source code. Here is the interpretation of Perkins' elements of understanding, in the case of programs:

- The *structure* is represented by the program's methods because they are the smallest block of instructions capable of having a purpose on their own.
- The *purpose* is the set of business-related functionalities implemented by a program (the goals of the program).
- The *argumentation* shows how the program's *structure* carries out these functionalities (*purpose*). Therefore, the argumentation is an explanation of the program's source code.

Taking all previous considerations into account, we complemented Biggerstaff et al.'s definition with Perkins' ideas to propose our own definition of program understanding: *"Program understanding, considered not to be a process but a process's outcome, is achieved when three different kinds of knowledge are acquired: (1) the structure of the program, (2) the business domain functionalities carried out by the program and (3) an explanation of the program's structure and behavior that justifies its ability to perform these functionalities."*

3 UNDERSTANDING ARTIFACT

Because of the big difference in granularity between the functionalities and the elements of the source code to be explained, we realized that the argumentation must provide a missing granularity step. In fact, since the *purpose* of a program explains **why** it has been built and its *structure* explains **how** the program satisfies its purposes, the missing step should answer the following questions:

- From the perspective of the purpose: **what** does the program need to do to perform each business functionality?
- From the perspective of the source code structure: **what** do source code methods accomplish that helps performing the business functionality?

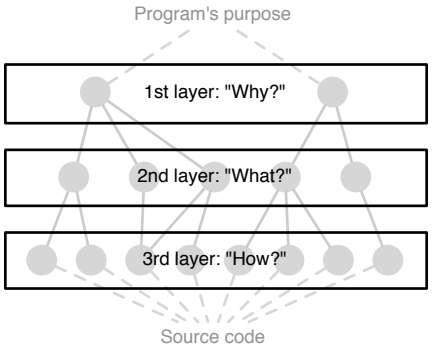

Figure 1: The understanding artifact and the hierarchy between its layers' elements

Then, our understanding model is structured in three layers. The two kinds of knowledge to be connected (*why* and *how*) are represented by the first and third layer respectively. The intermediary layer contains the knowledge connecting the elements between the first and third layers: the *argumentation*. These layers correspond each to a particular level of abstraction from the source code. Figure 1 shows the three layers of our understanding model (we refer to the elements of the second layer as "tasks" because they answer *what* the program must do). The connection between the layers are the following:

(1) A functionality in the 1st layer is decomposed into a sequence of tasks in the 2nd layer. This decomposition explains in business terms what the program needs to do to fulfill the functionality. This sequence of tasks is understandable by domain experts.
(2) A task in the 2nd layer is linked to the source code methods in the 3rd layer. These methods explain how the program carries out (implements) the task.

The tasks in the second layer are much more abstract than the methods in the source code (third layer). We must, therefore, find a way to close this gap. To this end, we introduced the notion of *manipulations* [4], representing atomic business domain information processing activities. Each task is carried out by a *manipulation* and the methods implement *manipulations*. Our formalization of the notion of manipulation was inspired by the use of verbs in natural language (NL) because business domain information processing activities can easily be transformed into simple declarative sentences. More specifically, the business domain processing action would be the verb of such a declarative sentence, and the business concept that is processed would be the sentence's direct object. Since the second layer's elements result from the decomposition of the functionalities into tasks, and since the tasks are carried out by manipulations, each of the functionalities is now mapped to a sequence of manipulations. Inversely, the program behavior is represented by the tasks' sequence which is the sequence of information manipulation the program is supposed to perform. Now we need a way to represent the order in which tasks must be carried out for each functionality. We chose to use the Business Process Model Notation (BPMN) because it is easy to understand for non-developers. Then, this is the formalism we have used for the second layer of our model:

(1) The description of the program's functionalities and the decomposition of the latter into BPMN models cannot be automated. It must be performed manually with the help of the users of the program and the domain experts. But, the most laborious part is to connect the BPMN tasks to the methods in the source code methods. This is what we automated, which is the key contribution of the paper.
(2) In summary, our model reconstructs the abstraction levels between the purpose of the program and the source code.

4 DYNAMIC ANALYSIS

To recover the connections between the second and third layers we relied on a dynamic analysis technique i.e. post-mortem execution trace (XT) analysis. In our research, an XT records the sequence of method calls as call trees. All method calls are made from within

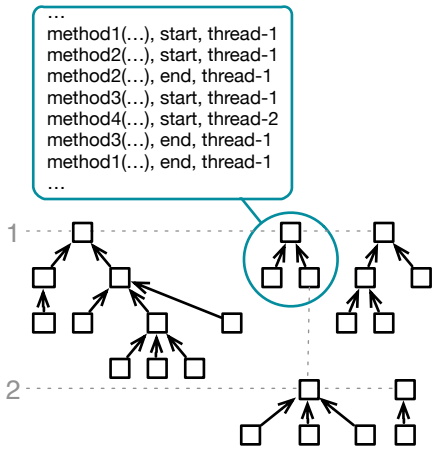

Figure 2: XT's graphical representation

other method calls, with the exception of methods called by the system itself (e.g. event listeners). Since there could be several threads in any program execution, any XT can be represented as a "forest" of method call trees. To visualize the hierarchies of calls in an XT we designed our own graphical representation. Figure 2 shows an example of such a representation: *squares* represent the method calls and *arrows* represent the callee-caller relation (oriented towards the caller). The topmost method calls in each tree (root) are those made by the system or by a method in another thread. These root methods are grouped into threads by a horizontal dotted line marked on the left with the thread number. The vertical dotted lines are used to connect the calls in different threads: they connect the method called in a new thread to the calling method in another thread.

4.1 BPMN Task Detection

4.1.1 Requirements. BPMN tasks and their sequential dependencies are modeled in the second layer of our artifacts. They represent the sequence of tasks required to perform the corresponding functionality. When we record the XT associated with a program functionality, the sequence of methods called must correspond one way or another to the sequence of tasks in the BPMN model. Now the problem to solve is to link the methods called to the task they implement. This is based on two clues:

- The kind of work carried out by a method in business terms. This is represented by the *manipulation* associated to the task (section 3).
- The moment in time the method is called in the XT, or more specifically the relative position of the method in the sequence of calls. This must correspond to the task whose position is comparable in the sequence of tasks in the BPMN model.

4.1.2 Definitions. Unlike the connections between the functionalities and the BPMN tasks, the connections between the BPMN tasks and the methods are not one-to-many but many-to-many. While the BPMN tasks might be implemented by more than one method, methods can also be part of the implementation of more

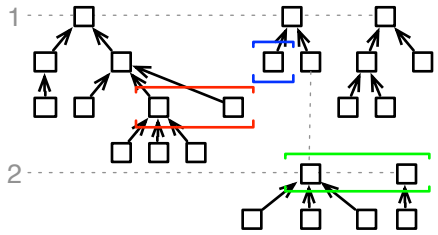

Figure 3: Graphical notation for segments

than one BPMN task. The first clue to sort-out this situation comes from the fact that each BPMN task is implemented as a *unit of work*: a set of contiguous information processing steps. Therefore, the corresponding caller and callee methods must be close in the XT. To represent such temporal proximity, we introduced the notion of *trace segment* (or *segment* for short) associated to each BPMN task. It is the part of the XT between the first method call and the last method call corresponding to this task in a given thread. Since the methods corresponding to a single BPMN task may span several threads, it may be associated to a *set* of segments. There must be at most one segment per thread for a given BPMN task. A segment may contain a single method call or the multiple contiguous method calls of the method call tree. Starting from the graphical representation introduced in Figure 2, we use a pair of vertical brackets to identify the *root* methods of each segment. The colored brackets in Figure 3 identify the root method calls in three segments (the methods called from these methods also belong to the segment).

Let us say that a BPMN task was carried out by the method calls in the blue and green segments in Figure 3. Then, the set containing those two segments would be the XT counterpart of the BPMN task. Moreover, let SH be the set of all sets of segments, including the empty set. Then, an XT *segmentation* is the attribution of one segment set in SH to each task in the BPMN. Formally, be T the set of BPMN tasks linked to a functionality. We define the segmentation as the function $SS = T \rightarrow SH$ which returns, for each task, the set of segments corresponding to the task.

4.1.3 Manipulation Detection. We have defined *manipulations* as pairs of an action and a business concept. A manipulation is identified in a method if each component in the pair is detected in it. Because of their different nature, actions and business concepts are implemented differently in code. An action represents how a program processes some business information. Then, the identification of an action in a method involves inspecting its source code to see if it implements the action. We perform this analysis by comparing the methods' source code to a record of "syntax clues" stored in our action ontology. This gives the possible implementations of each action. In [4] we introduced an action ontology as a formalization of generic actions. On the other hand, business concepts represent the business information that is processed. Their identification in the source code is based on the widely accepted hypothesis that business domain concepts materialize as program identifiers [1, 11, 21, 26]. Then, every concept in our domain ontology is assigned a set of terms used to identify the concept. The identifiers in a method's source code will be compared to these

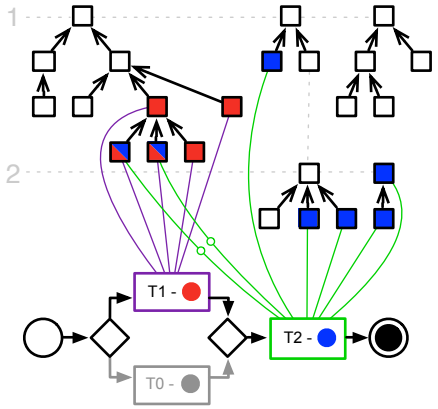

Figure 4: Example set of initial mappings produced by manipulation detection

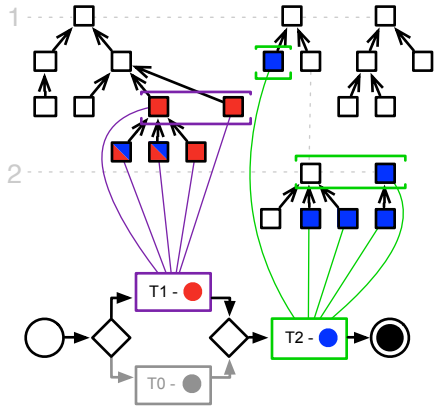

Figure 5: Example set of initial mappings filtered using a trace segmentation

terms and each match will be considered an evidence of the concept's presence.

4.1.4 Initial mapping. There are two steps to identify the methods corresponding to a BPMN task: first, find the manipulations and second, check the sequence of execution. Since several tasks may involve the same manipulation, the identification of the latter in a method is not enough to assign the method to the task. For example, let us assume that the XT showed in Figure 4 was recorded while executing a functionality represented by the BPMN model at the bottom of the figure. The colored dots in the tasks of the BPMN model represent manipulations assigned to these tasks. The color assigned to a square means that the manipulation is found in the corresponding method's source code. In particular, the red and blue manipulations are found simultaneously in two methods in the subtree on the left.

Such a first step may lead to a lot of false positives (wrong mapping between a task and a method). Indeed, since several methods could implement a given manipulation, they would be mapped to all BPMN tasks associated to that manipulation. However, the order of execution of the methods and the tasks should be similar. Therefore, the methods colored both blue and red in the figure cannot be mapped to T2 since they belong to the segment associated to T1 whose execution is completed when the methods associated to T2 are executed. The false mappings (false positive) are represented by the green connections marked with an empty circle in Figure 4.

4.1.5 Filtered Mappings. Since the segment sets in *SH* can be considered to be the XT counterpart of the BPMN tasks, we can exploit the information on the sequence of the tasks in the BPMN model to filter out the mappings. In other words, we will verify that the segments mapped to the tasks are in the same order as the tasks in the BPMN model. Under these constraints, the resulting mapping will represent the missing links between the second and third layers of our model. Figure 5 represent the same information as Figure 4 but the false mappings have been removed.

Now the problem is to build the correct segmentation automatically, i.e. to find the set of segments that truly identify the methods implementing the BPMN tasks. As a summary, there are two distinct representations of the program behavior that we need to compare:

(1) The sequence of the BPMN tasks that model, at the business level, the processing required to carry out functionality. We call it the "expected behavior".
(2) The sequence of method calls corresponding to the execution of the program when launching the computation of the functionality. We called it the "observed behavior".

5 AUTOMATION

5.1 Manipulation Detection

The two parts of a manipulation to be searched for in the source code are the business concept and the action performed on the concept.

5.1.1 Business Domain Concept Detection. To check for the presence of a business concept, we analyze the identifiers used in the methods' source code. To perform this search, every business concept in the domain ontology is assigned a set of the terms that, if found in a method, could be an evidence of the concept's presence in the method. Since we usually work with preexisting domain ontologies found on the web (we do not build our own from scratch) the set of strings we start from to find the terms are built from:

(1) The concept's URI's fragment identifier. A short string of characters that unambiguously identifies a resource within its ontology.
(2) The concept's label or name, which is provided by the `#label` data property in an RDF schema.
(3) The concept's attributes' URI fragment identifiers and their labels.

Next, we apply the appropriate word separation technique on the above strings by:

- Detecting camel-casing, dashes or underscores in identifiers.
- Detecting spaces and punctuation marks in NL strings.

After removing all stop-words from the resulting sets of words, we use the Snowball API, an algorithmic stemmer, to reduce inflected words to their roots. The union of the resulting sets constitutes the set of terms assigned to each concept. Next, the same process is applied to the identifiers found in the method's source code. A business concept is detected in a method if the concept's set of terms intersects with the set of stemmed identifiers of the method.

5.1.2 Action detection. The possible actions on the concepts are domain-independent. We represent them in an action ontology [4]. To detect an action in a method, we compare the method's source code to a set of syntax clues assigned to every action in the ontology. These syntax clues are programming language dependent. Then, there will be a set of syntax clues for each programming language considered. The actual specification of the syntax clues can be done in many different ways. In our experiments, Java was the target programming language and the specification of the syntax clues was done using entities from Eclipse's Java Development Tools (JDT), i.e. AST node visitors.

5.1.3 Recall Strategies. Achieving high recall rate during the manipulation detection is important to make sure that at least the correct segments are part of the set of mappings to be filtered out. Then, we implemented two strategies to increase the recall rate. First, we realized that the methods carrying out a particular task may delegate part of the work to the methods they call. Consequently, the first strategy is to explore the full call tree from each root method in a segment when searching for a specific manipulation. Second, we observed that a method might process some rather unspecific attribute of a concept to implement the manipulation. For example, let us consider a hotel reservation management system. When searching for the $(Update, Reservation)$ manipulation, a developer might have implemented an $(Update, Date)$ manipulation instead, if the reservation's date was the only thing that could be updated. But the "Date" attribute is rather common among concepts. Therefore, the identification of the concept "Reservation" is ambiguous. Hence, the second strategy is to define some semantic distance between manipulations, so that the manipulations that are "close" to the expected one will be retrieved. We started from the measure of semantic relatedness among concepts proposed by [16]. This computes the distance between two concepts in the ontology graph taking into account any ontology relation. But we changed the definition of maximum distance between two concepts to use our own. Then, the distance between two concepts in our ontologies is the weight of the shortest multi-relation path between the concepts, normalized by the maximum distance. From this development, we can now compute the distance between the manipulations from the distance computed among the actions concepts and the business concepts in their own ontologies. The distance between 2 manipulations is the average of the distance between their corresponding action and business concepts. When searching for a manipulation we will take into account all manipulations whose distance is inferior or equal to some threshold.

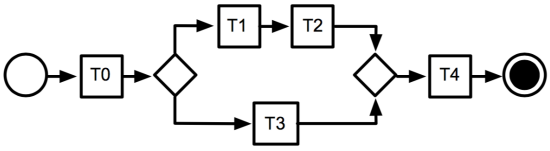

Figure 6: BPMN model (2nd layer)

5.2 Behavioral Comparison

Usually, BPMN models represent all the alternative task sequences to implement a given functionality. For example, in Figure 6, the model on the top shows that the sequence of tasks T0, T1, T2, T4 or T0, T3, T4 are both valid to implement the functionality. But in an actual execution of the program, only one of those sequences is executed. We refer to this executed sequence as the Execution Path (XP). The segmentation must then be associated to the tasks of the XP, not to the whole model which contains many alternatives. To identify what particular XP was executed in some run of the system, we must observe the behavior of the system. But the only observable behavior is represented by the views (screens) of the system, which are associated to specific tasks. The rest of the path must be deduced from the BPMN model. In the case the complete task sequence cannot be deduced unambiguously, we replace the unobservable tasks with the smallest part of the model containing all of them. The corresponding set of tasks will be represented as a single abstract task. For example, let us assume that T0 and T4 are the only observed tasks from the BPMN model in Figure 6, the XP would be: T0, G, T4. Where G is an abstract task representing the center of the model. But, if the task T2 would have been observed through the display of some associated view, then we would know the complete task sequence: T0, T1, T2, T4. To explore the segmentation space, we generate alternative segmentations and assess their relevance. The starting point is what we call the *full segmentation*: one segment per thread, each segment containing all the method call in the corresponding thread. Then we generate alternative ones by adding or removing methods from its segments. There are four operations: remove-left (RL), add-left (AL), remove-right (RR) and add-right (AR) that add or remove one method call (on the left or on the right) from the segment attributed to a specified task on a given thread. To evaluate the quality of the resulting segmentation (the fitness function), we use three heuristics:

- **HS:** compares the segmentation to the expected behavior.
- **HC:** compares the segmentation to the observed behavior.
- **HD:** verifies that the segments attributed to the tasks among different threads are close in time to each other, to ensure that the tasks do represent "cohesive" units of work.

To compare a segmentation to the expected behavior, we transform the XP as well as the segmentation into a directed graph of tasks. Then, HS is computed as the ratio of arcs in the graph of XP that are missing in the graph of the segmentation.

To compare the segmentation to the observed behavior, we assess the capacity of the former to predict the latter. To do so we build a contingency table and use Matthews' Correlation Coefficient (MCC) [15], a general prediction quality assessment, to compute a

value in $[-1, 1] \subset \mathbb{R}$ (where TN = True Negative, TP = True Positive, FN = False Negative, FP = False Positive):

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The principle underpinning HD is that, since BPMN tasks represent units of work, the system should also execute them as units. HD is computed as the average, over all the BPMN tasks involved in the XP, of the ratio of the XT events not attributed to a task in the interval defined by the first and last events attributed to the task. The final fitness function is computed as the median of the values returned by these heuristics.

6 EXPERIMENT

We tested our approach on JHotel [13], a small open source application for the room management of a hotel. To assess the quality of the recovered links between the methods and the tasks, we compared it with a gold standard: the mapping built manually. We decided to use JHotel as a case study since hostelry (JHotel's business domain) is well known and its functionalities are easy to understand. Another reason for our choice was the limited size of the system. JHotel is big enough to make its source code and XTs hard to process without a tool, but small enough for us to be able to manually build the mapping. We built a hostelry domain ontology by importing the following ontologies into a single one:

(1) **HDeO:** Hotel Domain Ontology developed in [25], which includes the concepts associated with hotels themselves, rooms, facilities and neighboring attractions or services.
(2) **Time Ontology:** An ontology of temporal concepts being developed by the W3C group [12] needed to describe dates and durations.
(3) **Person Ontology:** An ontology of persons, needed to model guest concepts [14].

We documented two functionalities for the case study: "Create new reservation" and "Check-in guest". The first functionality is more complex than the second because it involves a larger number of options and parameters to select. Consequently, it requires more BPMN tasks (14 vs. 5) and uses a larger variety of business domain concepts (5 vs. 1).

Table 1: Method call tree propagation results

	Check-in guest		New reservation	
	MCC	Recall	MCC	Recall
Simple	0.35	0.5	0.22	0.72
W/propagation	0.41	0.78	0.20	0.81

Table 1 shows the MCC and recall values computed for the mapping generated by (1) a simple manipulation detection and (2) by the manipulation detection enhanced with the first strategy explained in section 5.1.3 (search in the full call tree).

As we can see, this strategy is able to raise the recall up to approximatively 80% for both functionalities; additionally, it does it with just a slight negative impact on the MCC for one of the functionalities. The assessment of the second strategy, i.e. computing

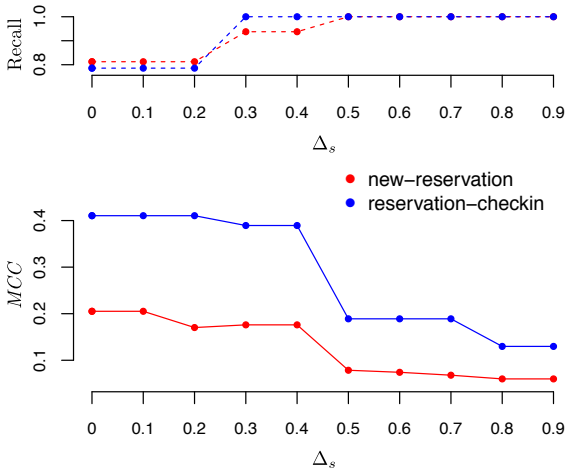

Figure 7: Impact of different Δ_S on the manipulation detection results

distances among manipulations (see section 5.1.3), requires us to decide on the threshold (Δ_S). Then, we tried 10 different values for Δ_S, spread uniformly between 0 and 0.9. The results of this experiment are shown in Figure 7. As we can see, this strategy is able to increase the recall value by up to 20%, starting from $\Delta_S = 0.3$. However, while the recall rate increases monotonically, reaching 100% for both functionalities at $\Delta_S = 0.5$, the MCC measure rules out $\Delta_S > 0.4$. Indeed, although this strategy has just a small negative effect on the MCC when $\Delta_S < 0.5$, the quality of the resulting mapping drops significantly at $\Delta_S = 0.5$ for both functionalities. Then, to limit the negative impact on the MCC while still benefiting from a higher recall rate, we used $\Delta_S = 0.3$ in the following evaluations.

In Figure 8, we show the histogram of the MCC evaluations of the mappings filtered using the segmentations returned by 100 executions of the algorithm for each functionality. The height of the bars represents the relative frequency with which the MCC evaluation for a segmentation returned by the algorithm fell in the bar's interval. The dashed line stands for the MCC of the *full segmentation* (see section 5.2) which is the starting point for the exploration of the segmentation space. The numbers on the top corners of each graph show the percentage of segmentations returned by the algorithm having an MCC evaluation falling on each side of the dashed line. The top-right percentage value is that of the segmentations found by the algorithm that are better than the starting point of the exploration. While the most likely MCC value (the highest bin) obtained by our algorithm is for both functionalities better than the full segmentation, the segmentations returned by the algorithm for the "Check-in guest" functionality were half the time worse than the full segmentation (blue top-left percentage value in Figure 8). However, the algorithm performs better for the "Create new reservation" functionality, finding a better segmentation 66% of the time (red top-right percentage). It seems that having a more complex BPMN model for a functionality makes the search for a good segmentation easier. We consider this to be plausible because having more tasks in the BPMN model leads to a greater

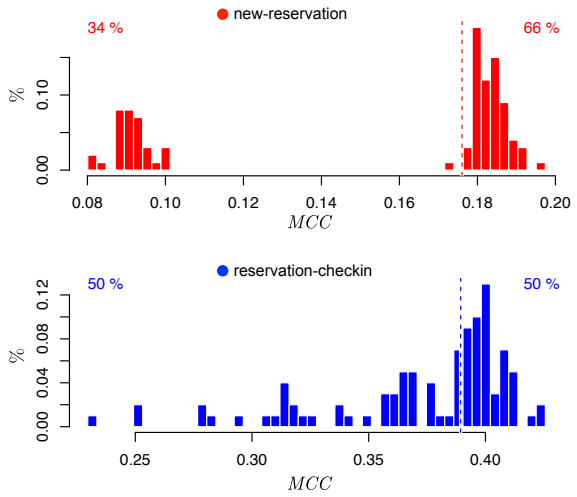

Figure 8: Results of the segmentation algorithm

diversity in the manipulations, which could make the observed behavior easier to segment. Indeed, an increase in the semantic distance between the manipulations used in a BPMN model makes the tasks more likely to be mapped to different methods, which in turn makes the latter easier to group together.

7 RELATED WORK

Besides their definition of program understanding, Biggerstaff et al. also introduced the term "concept assignment" as a general approach to tackling the understanding problem [7]. Concept assignment refers to the search and assignment of "human-oriented" concepts to the elements of the program code. Biggerstaff et al. explained that during program understanding, the software engineer must discover the reference to human-oriented concepts in the program's elements and interrelate them into a "human-oriented expression of computational intent," something similar to what Perkins called the *argumentation* part of understanding [20]. What Biggerstaff et al. called "concepts" could be understood in multiple ways. But most of the research in concept assignment focuses on the search for reference to domain concepts in the source code through identifiers. Recently, works in software comprehension tend to concentrate on the building of models of software understanding. This represents a change in the program understanding community from the earlier days, where tools to analyze code were more frequent in the literature. For example, Benomar et al. [6] noted that research on software comprehension was generally split into two distinct areas: program design understanding and program evolution understanding. Hence, they proposed a unified model encompassing the time-dimension to address both areas: the dynamic aspects of software execution and the evolution over time of the software. Nosal et al. [19] observed through controlled experiments that the whole understanding process is hypothesis-based and consists in matching elements found in source-code (solution domain) to the software requirements (problem domain). The authors claim that the understander's current knowledge and prior experiences about the problem and the solution domains is the base on which mapping hypotheses are constructed to recover the mental model used by the original developers. But to find works on tools and techniques to actually build these models, one must go a bit earlier in the literature. For example, identifiers and comments were analyzed by Anquetil [1] to study whether they could be used to recover traceability links between the domain concepts and the implementation components. Haiduc and Marcus [11] analyzed the likelihood for different developers to use the same terms to refer to the same concepts. They found that, although in general the probability of two people choosing the same term to refer to the same concept is around 20%, in the case of developers, the probability is as high as 63%. This shows that there is an important level of agreement in the choice of terms made by developers. Therefore, it makes sense to use a formal representation of the concepts of the domain, e.g., an ontology, as a source of information during software reengineering activities. However, the explicit representation of ontologies in software engineering emerged only recently. Djuric and Devedzic [10] explain that this might have been the case because the efforts in developing ontology languages and tools were focused on developing the Semantic Web rather than on programming or software engineering practices. Ratiu and Deienbck [22] describe an ontology-mapping approach for the concept location in programs. They started from two ontology-like graphs: first, they used WordNet, the famous lexical database, second, they used a graph representation of the identifiers found in the source code. Then, the authors applied their mapping procedure on JFreeChart2, an open source Java library for drawing charts. They were able to map 20% of the identifiers in the source code to a WordNet concept. However, because of the imprecise definition of what is meant by "program understanding" in the literature, it is generally not clear to what extent the published works contribute to this research goal. For example, Asadi, F. et al. [2] consider concept location as the detection of cohesive and decoupled fragments of an execution trace using Latent Semantic Indexing (LSI). But it is not clear how the concepts attached to the fragments would help with program understanding. Indeed, the "concepts" in this context are sets of terms that do not necessarily represent a single notion or a single concept in the sense of Biggerstaff et al. But Medini, S. et al. [17] seem to have identified the problem since they introduced a technique to label the fragments of the execution trace. Although this technique does improve the quality of fragments in terms of cohesion and decoupling, it fails to address the main issue.

8 CONCLUSION

Although the usefulness of our model in helping developers understand programs has already been shown elsewhere [5], the problem that remained to be solved was its automatic generation. Indeed, the model is too complex to produce manually and the hardest part is by far the production of the mapping between the BPMN tasks and their implementation methods. In this article, we identified two techniques to automate this mapping. First, we introduced the idea of *manipulation* to represent the information processing activities carried out by the BPMN task. Second, we leveraged the correspondence between the position of a task in the BPMN model and the sequence of execution of the corresponding method in the execution trace. From this background, we presented the way we

implemented those ideas and specifically the two strategies aimed at increasing the recall of the manipulation detection: method call tree propagation and a distance threshold between manipulations. Using these strategies, we achieved high recall rates in our experiments: 100% for the "Check-in guest" functionality and 93% for the "Create new reservation" functionality. Then, we used a fitness function in a hill-climbing search algorithm to filter out the results. The latter are encouraging but mixed. Indeed, while the mapping obtained for a rather complex BPMN model was good ("Create new reservation" functionality), it was not very good for a simple one ("Check-in guest" functionality). This suggests that having a more complex BPMN model for a functionality makes the search for a good segmentation easier.

Future work should therefore explore this finding. In particular we may focus on studying the segmentation space because its properties could be key to the identification of more effective search algorithms. Finally, it must also be highlighted that the performance of our approach is comparable to state of the art techniques as reported in the literature. We expected nonetheless some extra performance that we unfortunately did not reach so far. We may however improve the technique in the proposed future work.

REFERENCES

[1] Nicolas Anquetil. 2001. Characterizing the informal knowledge contained in systems. In *The 8th Working Conference on Reverse Engineering*. Stuttgart, Germany, 166–175.

[2] Fatemeh Asadi, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2010. A Heuristic-Based Approach to Identify Concepts in Execution Traces. In *14th European Conference on Software Maintenance and Reengineering*. 31–40.

[3] Jonathan Baron. 2009. *Thinking and Deciding* (4th ed.). Cambridge University Press, Cambridge.

[4] Javier Belmonte and Philippe Dugerdil. 2010. Using domain ontologies in a dynamic analysis for program comprehension. In *2nd International Workshop on Ontology-Driven Software Engineering*. ACM, Reno, NV, USA.

[5] Javier Belmonte, Philippe Dugerdil, and Ashish Agrawal. 2014. A three-layer model of source code comprehension. In *7th India Software Engineering Conference*. ACM Press, Chennai, India.

[6] Omar Benomar, Houari A. Sahraoui, and Pierre Poulin. 2015. A Unified Framework for the Comprehension of Software's Time.. In *37th IEEE International Conference on Software Engineering*. Florence, Italy.

[7] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1993. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*.

[8] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E Webster. 1994. Program understanding and the concept assignment problem. *Commun. ACM* 37, 5 (May 1994), 72–82.

[9] Richard Clayton, Spencer Rugaber, and Linda Wills. 1998. On the knowledge required to understand a program. In *5th Working Conference on Reverse Engineering*. 69–78.

[10] Dragan Djuric and Vladan Devedzic. 2011. Incorporating the Ontology Paradigm Into Software Engineering: Enhancing Domain-Driven Programming in Clojure/Java. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 42, 1 (May 2011), 3–14.

[11] Sonia Haiduc and Andrian Marcus. 2008. On the Use of Domain Terms in Source Code. In *The 16th IEEE International Conference on Program Comprehension*. 113–122.

[12] Jerry .R. Hobbs and Feng Pan. 2006. Time Ontology in OWL. (2006). Retrieved December 12, 2017 from http://www.w3.org/TR/2006/WD-owl-time-20060927/

[13] JSoft-Munich. 2004. JHotel. (June 2004). Retrieved December 12, 2017 from https://sourceforge.net/projects/jhotel/

[14] D. King and M. Hall. 2003. Person Ontology. (2003). Retrieved September 2, 2010 from http://orlando.drc.com/semanticweb/daml/ontology/person/person-ont/

[15] B. W. Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405, 2 (Oct. 1975), 442–451.

[16] Laurent Mazuel and Nicolas Sabouret. 2008. Semantic Relatedness Measure Using Object Properties in an Ontology. In *The Semantic Web - ISWC 2008*. Springer Berlin Heidelberg, Berlin, Heidelberg, 681–694.

[17] Soumaya Medini, Giuliano Antoniol, Yann-Gaël Guéhéneuc, Massimiliano Di Penta, and Paolo Tonella. 2012. SCAN: An Approach to Label and Relate Execution Trace Segments. In *19th Working Conference on Reverse Engineering*. 135–144.

[18] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. 1993. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *3rd Conference of the IBM Centre for Advanced Studies*. Toronto, ON, Canada.

[19] Milan Nosal and Jaroslav Poruban. 2015. Program comprehension with four-layered mental model. In *13th International Conference on Engineering of Modern Electric Systems*. Oradea, Romania.

[20] David N. Perkins. 1986. Knowledge As Design. In *Knowledge As Design*. 1–19.

[21] Daniel Ratiu. 2009. Reverse Engineering Domain Models from Source Code. In *International Workshop on Reverse Engineering Models from Software Artifacts*.

[22] Daniel Ratiu and Florian Deißenböck. 2006. How Programs Represent Reality (and how they don't). In *13th Working Conference on Reverse Engineering*. 83–92.

[23] Margaret-Anne D. Storey. 2005. Theories, methods and tools in program comprehension: past, present and future. *13th International Workshop on Program Comprehension* (2005), 181–191.

[24] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (Aug. 1995), 44–55.

[25] Donghee Yoo, Gunwoo Kim, and Yongmoo Suh. 2009. Hotel-Domain Ontology for a Semantic Hotel Search System. *Information Technology & Tourism* 11, 18 (Jan. 2009), 67–84.

[26] Hong Zhou, Feng Chen, and Hongji Yang. 2008. Developing Application Specific Ontology for Program Comprehension by Combining Domain Ontology with Code Ontology. In *8th International Conference on Quality Software*. 225–234.