Noname manuscript No.
(will be inserted by the editor)

An Approach for Bug Localization in Models Using Two Levels: Model and Metamodel

Lorena Arcega · Jaime font · Øystein Haugen · Carlos Cetina

Received: date / Accepted: date

Abstract Bug Localization is a common task in Software Engineering, especially when maintaining and evolving software products. This paper introduces a Bug Localization approach that, in contrast to existing source-code approaches, takes advantage of domain information found in the model and the metamodel. Throughout this paper, we present an approach for bug localization in models (BLiM2) that applies the source code ideas for bug localization (textual similarity to the bug description and the Defect Localization Principle) and takes advantage of the domain information from the model and the metamodel. We evaluated our approach in BSH, a real-world industrial case study in the induction hob domain measuring the results in terms of recall, precision, the combination of both the F-measure and the Matthews Correlation Coefficient (MCC). Our study shows that our BLiM2 approach, which combines information from the model and the metamodel for the textual similarity and differentiates between the timespan from the model and metamodel, provides the best results in this work. We also performed a statistical analysis to provide evidence of the significance of the results. The values obtained shows that exist significant differences in the performance of the best BLiM2 approach with the approach used by our industrial partner. Finally, the effect size statistics reveals that the best BLiM2 approach obtain better results in the 78% of the times in the worst case.

Keywords Bug Localization · Model Driven Engineering · Reverse Engineering

Lorena Arcega · Jaime Font
Universidad San Jorge. Escuela de Arquitectura y Tecnologia. Zaragoza. Spain
University of Oslo. Department of Informatics, Oslo, Norway
E-mail: larcega,jfont}@usj.es

Øystein Haugen
Østfold University College. Faculty of Computer Science. Halden, Norway
E-mail: oystein.haugen@hiof.no

Carlos Cetina
Universidad San Jorge. Escuela de Arquitectura y Tecnologia. Zaragoza. Spain
E-mail: ccetina@usj.es

1 Introduction

Nowadays, software exists in almost everything. This trend has been accompanied by a high increase in the scale and the complexity of software. Unfortunately, this is also accompanied by more software bugs. Hence, software maintenance is becoming increasingly important. Lehman et al. [33] pointed out that up to 80% of the lifetime of a system is spent on maintenance and evolution activities. Software maintainers spend from 50% up to almost 90% of their time trying to understand a program in order to make changes correctly.

Bug Localization is one of the most important and common activities performed by developers during software maintenance and evolution. Bug Localization aims to identify the location in the artifact that is pertinent to a software fault. A recent survey [61] reveals that none of the Bug Localization approaches take into account models as the source of the bugs. In the model paradigm, models can play several roles in software development: diagrams for analysis, can be reverse-engineered from source code, or can be used for code generation. In this work, we focus on models for code generation. When models are used for code generation, addressing bugs at the model level must not be neglected.

In the case of Bug Localization in source code, approaches are based mainly on information retrieval [51,65]. Some works [62,55] also take into account the Defect Localization Principle. This principle is based on the observation that the most recent modifications to a project are most likely the cause of future bugs [23, 66]. Taking into account recent modification may lead to finding relevant locations that are the cause of a bug [55].

In this paper, we propose an approach for Bug Localization in Models. This approach enables us to evaluate to what extent the ideas that have been successfully applied in source code for bug localization (textual similarity to the bug description and the Defect Localization Principle) also work for models. Our approach takes into account a property of the models that is not present in source code, namely, the domain information embedded in models and metamodels.

In our approach, information retrieval is used to measure the similarity of model fragments with the description of the bug report that we want to locate. Model fragments are formed by model elements, and each model element has an associated modification time. The Defect Localization Principle is measured through the timespan weighting that assesses the model fragment solutions using the timespan of the model modifications.

We materialize our Bug Localization approach as a Multi-Objective Evolutionary Algorithm that uses both the similarity to the bug report and the timespan weighting as fitness functions. Our approach is, in fact, a family of approaches. We can change from one of our approach to another by changing how we use the information of the models and the metamodels with which we measure the similarity and the timespan. As a result, software engineers obtain a ranked list of model fragments from the model, which is intended to identify the parts of the model that are relevant to the bug.

We have applied our approach to the product models from BSH, one of the largest manufacturers of home appliances in Europe. Its induction division has been producing induction hobs under the brands of Bosch and Siemens for the last 15 years. The firmware that controls the induction hobs is specified by means of a Domain Specific Language (IHDSL). The different configurations of the induction hobs are managed following a model-based Software Product Line (SPL) approach that uses Common Variability Language (CVL) [24] to configure their models. The firmware of their products is generated from the IHDSL models.

In our evaluation, we compare our Bug Localization approach, which uses model and metamodel information (BLiM2), with a baseline approach. The baseline approach is used by BSH for bug localization. We apply both the BLiM2 approach and the baseline to the product family of BSH. They provided us with documentation about bugs. For each one of the bugs, the documentation provided a bug description and the localization of the bug. Taking the bug descriptions and the product family as input, we measure the results using the standard measurements accepted by the software engineering community: recall, precision, the combination of both the F-measure, and the Matthews Correlation Coefficient (MCC) [52, 40] using the location of the bugs as oracle.

The variant BLiM2-3OT of our approach achieves the best results. It has three objectives in the fitness function: one that combines information from the model and the metamodel for text similarity; one that takes into account the modification timespan of the model; and one that takes into account the modification timespan of the metamodel. The results in terms of recall, precision, and MCC, on average are 90.30%, 79.66%, and 0.83, respectively. Finally, we perform a statistical analysis on the results in order to provide quantitative evidence of the impact of both the BLiM2 approach and the baseline approach and to show that this impact is significant.

The remainder of the paper is structured as follows. In Section 2, we present the Domain Specific Language used by our industrial partner and the differences between feature localization and bug localization. In Section 3, we describe our BLiM2 approach. In Section 4, we evaluate the application of our approach in BSH. In Section 5, we examine the related work of the area. Finally, we present our conclusions in Section 6.

2 Background

The running example and the evaluation in this paper are performed using the products of our industrial partner, BSH. In this section, we present the Domain Specific Language (DSL) that is used by BSH to formalize their products, which is called IHDSL. The Common Variability Language (CVL) is also presented. CVL is the language used by our approach to formalize the model fragments.

The newest Induction Hobs (IHs) feature full cooking surfaces, where dynamic heating areas are automatically generated and activated or deactivated depending on the shape, size, and position of the cookware placed on the top. In addition, there has been an increase in the type of feedback provided to the user while cooking. All of these changes have been made possible at the expense of increasing the complexity of the software behind IHs.

The Domain Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 meta-classes, 47 references among them, and more than 180 properties. To gain legibility and due to intellectual property right concerns, in this section, we show a simplified subset of the IHDSL (see Fig. 1, IHDSL Metamodel and IHDSL Syntax). However, the evaluation was performed using the full IHDSL that is used in BSH.

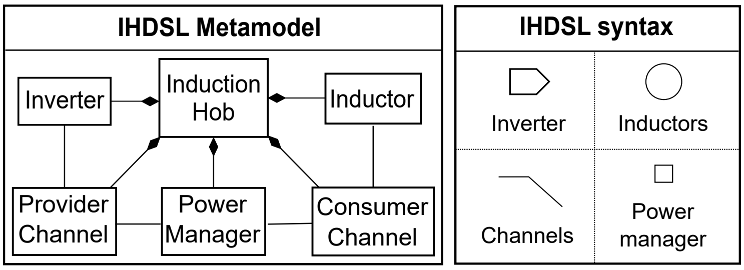


Fig. 1 IHDSL Metamodel

Inverters are in charge of transforming the input electric supply to match the specific requirements of the IH. Then, the energy is transferred to the inductors through the channels. There can be several alternative channels, which enable different heating strategies depending on the cookware placed on top of the IH at run-time. The path followed by the energy through the channels is controlled by the power manager. Inductors are the elements where the energy is transformed into an electromagnetic field.

The Product Model in Fig. 2 depicts an example of a product model that is specified with the IHDSL. The product model contains four inverters that are used to power two different inductors. The upper inductor is powered by a single inverter while the lower inductor is powered by the combination of three different inverters. Power managers acts as hubs to perform the connection between the inverters and the inductors.

To formalize the solution of our approach as model fragments, we use Common Variability Language (CVL) [24,56], due to its capabilities to formalize a set of model elements as a model fragment. The Model Fragment in Fig. 2 shows an example of a model fragment of the product model (the Product Model in Fig. 2). The model fragment includes the three inverters (in charge of powering the lower inductor), the three channels, and the power manager that is used to aggregate and manage the power provided by those inverters. Then, the solution of our approach is formalized by means of CVL and showed to the engineers.

In addition, to address the evolution of the metamodel, our industrial partner uses the Variable MetaModel strategy (VMM) [17]. This strategy has achieved

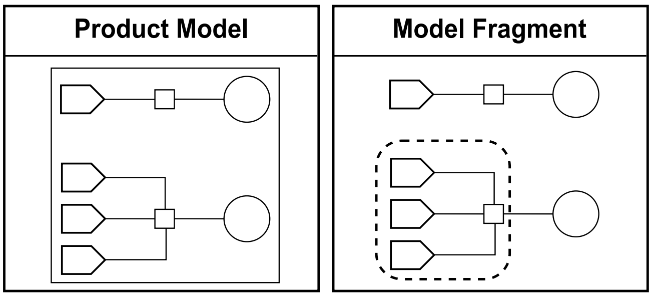


Fig. 2 IHDSL syntax, product model, and model fragment formalization

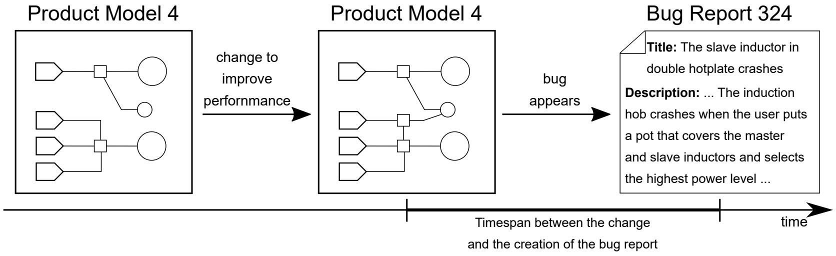

Product Model 4
change to
improve
performnance
Product Model 4
bug
appears
Bug Report 324
Title: The slave inductor in double hotplate crashes
Description: ... The induction hob crashes when the user puts a pot that covers the master and slave inductors and selects the highest power level ...
Timespan between the change and the creation of the bug report
time

Fig. 3 Example of a bug

better results than the migration strategy in domains like that of our industrial partner, BSH, in terms of indirection, automation, and trust leak.

2.1 Differences between feature and bug localization

In software systems, a feature represents a functionality that is defined by requirements and is accessible to developers and users. Software maintenance and evolution involves adding new features to programs, improving existing functionalities, and removing bugs, which is analogous to removing unwanted functionalities [15].

Bug localization is a specialization of feature localization. In the end, the software engineer obtains a piece of software that is in charge of some functionality in the system with either of the approaches. However, if we want to perform bug localization, we must take into account different particularities than if we want to perform feature localization. For example, the Defect Localization Principle can be applied to bug localization but not to feature localization.

Our previous approach [19] was developed for feature localization. It uses an evolutionary algorithm that iterates through the models of the system and assesses each one of the possible solutions. In the end, the software engineer obtains an ordered set of model fragments that fits the feature.

In this work, we adapt our feature localization approach [19] to obtain better solutions in bug localization. We use the same operations to iterate through the models; however, this approach has a new fitness function. The feature localization approach assesses each model fragment regarding textual similarity with the feature description. In addition to the textual similarity, this approach adds a fitness function that is specific for use with bugs. This fitness function exploits the timespan since the last modification.

Another difference is that our BLiM2 approach takes into account a property that is specific to the models. It uses the domain information embedded in the model and the metamodel. In other words, to assess each model fragment, our approach takes information from both, model and metamodel (text for measure textual similarity and timespan for measure the last modification). Nevertheless, the use of the domain information embedded in the metamodel is not specific for bug localization, it could be applied to feature localization.

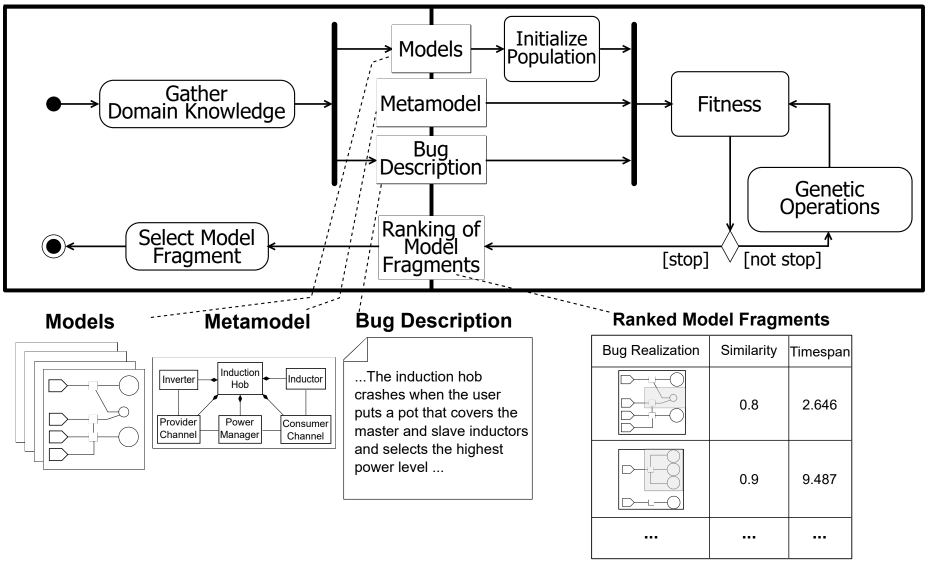
In this way, we convert the generic feature localization approach into a bug localization approach. This approach is a specialization of our previous feature localization approach. With our previous feature localization approach, users can locate features and bugs. In fact, the software engineers of BSH use our feature localization approach to locate bugs. The new specific bug localization approach exploits the timespan dimension of the last modifications, which is only relevant in bug localization.

3 The BLiM2 Approach

Fig. 3 shows an example of the use of our approach. The product model that appears on the left of the timeline is modified to make an improvement in the performance of its inductors. As a result, another product model is generated. This product model has a new power manager that connects the small inductor with the rest of the inverters. After some time of use, a bug appears and a bug report is generated. In the reminder of this paper, the timespan between the change and the creation of the bug report is called the modification timespan.

We materialize our BLiM2 approach as a Multi-Objective Evolutionary Algorithm that uses both, the similarity to the bug description and the modification timespan. The use of a multi-objective algorithm allows to show the results of both objectives (similarity and modification timespan). The effectiveness of each objective is different for each bug localization. Sometimes the similarity will be more successful to find the model fragment that contains the bug and sometimes the timespan is the more successful.

The objective of Bug Localization in Models is to obtain a set of model fragments from a given set of models that may correspond to a specific bug description being provided by the user of the approach. To do this, the approach receives a set of models and relies on an evolutionary algorithm that iterates a set of model fragments and evolves them using genetic operations. The evolutionary algorithm



...The induction hob crashes when the user puts a pot that covers the master and slave inductors and selects the highest power level ...

Bug Realization	Similarity	Timespan
	0.8	2.646
	0.9	9.487
...	...	...

Fig. 4 Overview of the Bug Localization Approach in Models: BLiM2

is guided by a fitness operation. As output, the approach provides a list of model fragments that should contain the bug. The overview of the process is shown in Fig. 4.

The left part of Fig. 4 shows the inputs used in our approach. The input is composed of a set of models, a metamodel, and a bug description:

- A set of product models that contain the target bug. The software engineer selects a set of product models from the entire family of products that contain the bug to be located.
- A metamodel to which the models of the product family conform.
- A bug description of the target bug, using natural language. Typically, these descriptions come from textual documentation of a bug report. Therefore, the query will include some domain specific terms that are similar to those used when specifying the product models. The knowledge of the engineers about the domain and the product models will be useful for selecting the description from the bug report.

The right part of Fig. 4 shows the main steps of our approach.

- The **initialize population** step calculates an initial set of model fragments from the input set of product models. This initial set of model fragments is randomly extracted from the product models. This is a common practice in evolutionary computation; as an alternative, seeds (fragments of a model chosen manually) can be proportionated in order to optimize the population, although that is out of the scope of this work.
- The **genetic operations** generate the new generation of model fragments. First, a selection operation selects the model fragments that will be used as parents of the new model fragments. The fitness values (see Section 3.2) are used to ensure that the best model fragments are chosen as parents. Then, a crossover operation mixes the model elements of the two parents into a new model fragment. Finally, a mutation operation introduces variations in the new model fragment (by adding or removing model elements) in hopes that the new model fragment achieves better fitness values than its parents.
- The **fitness** step assigns values that assess how good each model fragment is. The values will take into account the following terms:
  - Bug description: The more terms shared between the bug description and the properties of the model fragment (from the model and the metamodel), the higher this fitness value.
  - Modification timespan: The more recent the modification time, the higher this fitness value is. The timespan is calculated based on the difference between the last modification of a model element (in the model or in the metamodel) and the day on which the bug was discovered (see Fig. 3).

The output of BLiM2 (see Fig. 4) is an ordered set of model fragments that contains the target bug. The software engineer obtains this set of model fragments, which is intended to identify the parts of the model that are relevant to the bug. To do so, the engineer can order the ranking following different criteria depending on the variant of the approach used, such as the similarity to the bug description, and the most recent model fragment modifications.

Overall, the aim of the approach is to find the most relevant model fragment that contains the bug described by the bug report. To do so, the approach performs

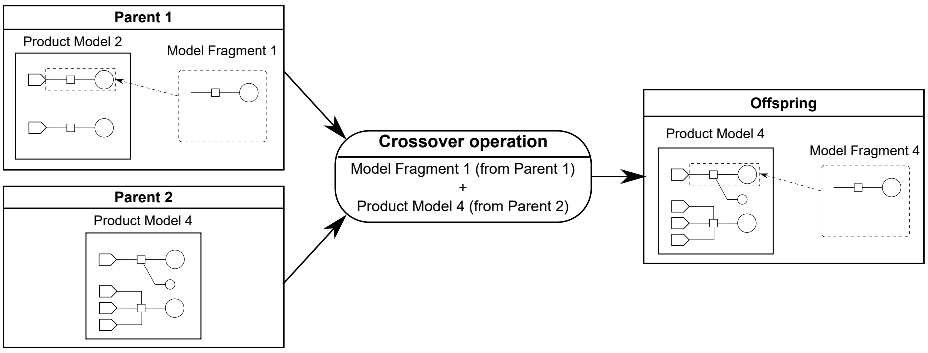


Fig. 5 Crossover Operation

a search guided by a fitness function. This search is done among the different model fragments (previously obtained applying mutation and crossover operations) that could contain the bug description. The fitness function will assign values based on the similarity with the textual description and the most recent model modifications of that model fragment.

The following sections describe the genetic operations of BLiM2 to generate new model fragments and how the fitness of each model fragment is determined in terms of similarity to the bug description, and the time of model modifications.

3.1 Genetic Operations of the BLiM2 Approach

The generation of subsequent model fragment sets is performed by applying genetic operators that we have adapted to work on model fragments in a previous work [19]. In other words, new fragments based on the existing ones are generated through the use of two genetic operators: the crossover operator and the mutation operator. These two operations are summarized in the following subsections, but the details are in [19].

3.1.1 Crossover

In our encoding, there are two elements that can be mapped across the different model fragments: the model fragment and the referenced product model. Therefore, our crossover operation will take the model fragment from the first parent and the product model from the second parent, generating a new model fragment that contains elements from both parents, thus preserving the basic mechanics of the crossover operation.

To achieve the above, our crossover operation is based on model comparisons. Fig. 5 shows an example of the application of the crossover operation on model fragments. First, we select the model fragment from the first parent. Then, we select the product model from the second parent. The model fragment (from the first parent) is then compared with the product model (from the second parent). If the comparison finds the model fragment in the product model, the operation creates a new model fragment with the model fragment taken from the first parent

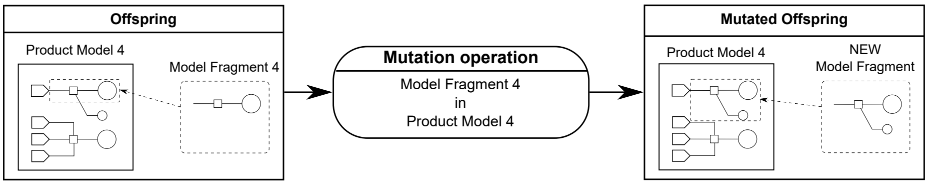


Fig. 6 Mutation Operation

but referencing the product model from the second parent. In the case that the comparison does not find a similar element, the crossover will return the first parent unchanged.

This operation enables the search space to be expanded to a different product model, i.e., both model fragments (the one from the first parent and the one from the new model fragment) will be the same. However, since each of them is referencing a different product model, they will mutate differently and provide different model fragments in further generations.

3.1.2 Mutation

Fig. 6 shows an example of our mutation for model fragments. Each model fragment is associated with a product model, and the model fragment mutates in the context of its associated product model. In other words, the model fragment will gain or drop some elements, but the resulting model fragment will still be part of the referenced product model. The mutation possibilities of a given model fragment are driven by its associated product model.

To perform the mutation, the type of mutation that will occur (either the addition or removal of elements) is decided randomly:

- **Subtractive Mutation:** This kind of mutation randomly removes some elements from the model fragment. The only constraint is that the elements be selected from the edges of the model fragment (they are connected with a single element). Therefore, the resulting model fragment is still connected (we can navigate from any element to any other element through the connections between the elements), and it is not split into several isolated groups of elements. Since the resulting model fragment is a subset of the original model fragment and the original was present in the referenced product model, the resulting product model will always be present in the referenced product model.
- **Additive Mutation:** This kind of mutation randomly adds some elements to the model fragment. The only constraint is that the resulting model fragment be part of the referenced product model. To achieve this, the boundaries of the model fragment with the rest of the product model are identified and then a random element from the boundary is added to the resulting model fragment. By doing so, the mutated model fragment will be part of the referenced product model.

Both, crossover and mutation operations are designed to produce individuals, by selecting subset of existing models to conform the new model fragment. The focus of this paper is generating a subset of model elements that are relevant to the bug that the domain expert wants to locate. The resultant model fragments are then presented to a domain expert for his assessment of its relevance for finding the bugs.

As a result, a new model fragment is created, but it still references the same product model. In other words, the model fragment represents another possible solution that can contain the bug (that is present in the product model) for the specific bug being located.

3.2 Fitness of the BLiM2 Approach

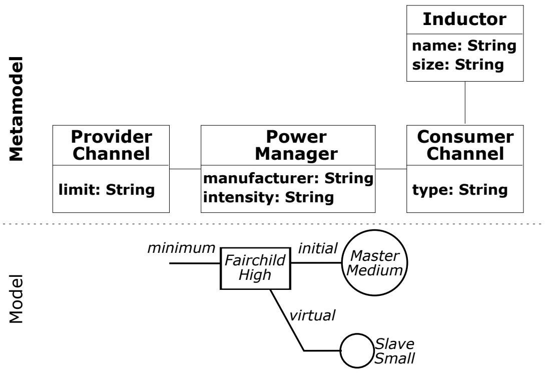
In evolutionary algorithms, the fitness step is used to assess the different degrees of adaptation to the environment that different model fragments have. Following this idea, our fitness step is used to determine the suitability of each model fragment to the problem. The input of this step is a set of model fragments, and the output produced is a set where each model fragment has been assigned with two fitness values: the similarity to the feature description, and the timespan to the most recent model fragment modifications.

3.2.1 Model Fragment Similarity to the Bug Description

To assess the relevance of each model fragment in relation to the bug description provided by the user, we apply methods based on Information Retrieval (IR) techniques. There are many popular IR techniques such as Vector Space Model (VSM) [53], Latent Semantic Indexing (LSI) [14] or Latent Dirichlet Allocation (LDA) [10]. The research findings are ambiguous and contradictory about which technique provides the best performance [57].

Based on our previous experience [6,19], we apply Latent Semantic Indexing (LSI) to analyze the relationships between the description of the bug provided by the user and the model fragments. Besides that LSI provides good results when applied to source code [50,35,46], a recent work reveals that LSI performs better when applied to text [48]. Product models are representations at a higher abstraction level than the source code, and the language used to build them is closer to the bug description language; similar to text.

LSI constructs vector representations of a query and a corpus of text documents by encoding them as a term by document co-occurrence matrix, (i.e., a matrix where each row corresponds to terms and each column corresponds to documents, with the last column corresponding to the query). Each cell holds the number of occurrences of a term (row) inside a document or the query (column). Fig. 7 shows the term extraction from a model fragment. The text of the document corresponds to the names and values of the properties and methods of each model fragment from the model and the metamodel, see Fig. 7. The metamodel provides the names of classes, attributes, and methods, while the model provides the values of these attributes and methods. The left part of Fig. 7 shows a model fragment with its corresponding metamodel fragment. The right part of Fig. 7 shows the terms extracted, the number of occurrences, and the source of the term (which comes from the model or from the metamodel). For example the term channel, appears 2 times in the fragment and the term channel comes from the metamodel (see the second column of the table in Fig. 7).



Term	Occurrences	Source
provider	1	Metamodel
channel	2	Metamodel
type	1	Metamodel
limit	1	Metamodel
manager	1	Metamodel
intensity	1	Metamodel
size	1	Metamodel
name	1	Metamodel
master	1	Model
slave	1	Model
high	1	Model
...	...	...
small	1	Model

Fig. 7 Terms extraction from a model fragment and the corresponding metamodel

Model Fragments

Terms	MF1	MF2	MF3	MF4	...	MFn	Query
size	6	12	9	6	...	12	6
provider	12	6	4	0	...	24	10
coil	36	30	0	0	...	0	0
small	24	12	2	0	...	6	4
intensity	0	8	10	2	...	0	6
high	6	0	2	0	...	6	8
...	...	...	...	...	...	...	0
name	8	2	1	12	...	3	0

Fig. 8 LSI applied to a model fragment

In our work, the documents are model fragments, i.e., a document of text is generated from each of the model fragments. The query is constructed from the terms that appear in the bug description. If the textual terms used for the model and the bug description differ too much, the LSI will not work. Therefore, the text from the documents (model fragments) and the text from the query (bug description) are homogenized by applying well-known Natural Language Processing techniques (tokenizing, Parts-of-Speech Tagging, and Lemmatizing) to eventually reduce this gap. If the languages used differ too much, other techniques such as manual annotation of the model elements could be applied at the expense of increasing the effort.

The union of all the keywords extracted from the documents (model fragments) and from the query (bug description) are the terms (rows) used by our LSI fitness (see Fig. 8). Each column is one of the model fragments; for example, the column that is shaded in grey in Fig. 8 is the one that corresponds to the model fragment from Fig. 7. The last column is the query obtained from the bug description of the

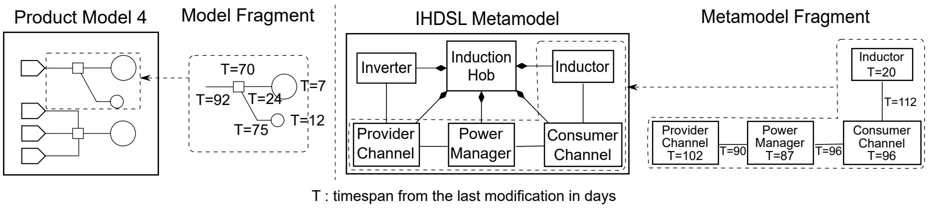


Fig. 9 Timespan of the modifications of the model elements of a fragment

user. Each row is one of the terms extracted from the corpuses of text composed by all of the model fragments and the query itself. Each cell has the number of occurrences of each of the terms in the model fragments.

Once the matrix is built (see Fig. 7), we normalize and decompose it into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [31]. One vector that represents the latent semantics of the document is obtained for each model fragment and the query. Finally, the similarities between the query and each model fragment are calculated as the cosine between the two vectors. The fitness value that is given to each model fragment is the one that we obtain when we calculate the similarity, obtaining values between -1 and 1.

To make the variants of our approach, we can take the union of the terms from the model and the metamodel (as shown in the example), or we can obtain different co-occurrence matrices. One co-occurrence matrix from the terms of the metamodel and another co-occurrence matrix from the terms of the model. This way, we will obtain two fitness values instead of one.

3.2.2 The Most Recent Model Modification

To apply the Defect Localization Principle, we measure the timespan to the last modifications of the model. As the modifications affect the model elements, each model element has its own modification timespan. Thus, each model element has a constant value of modification timespan during the execution of the algorithm, but different model elements have different modification timespan values.

The modifications of the model over time are considered when extracting the most relevant model from the target bug (the time difference between the last modification of a model element and the usage day). A recently modified model element (i.e., a short timespan) has a lower timespan value than another model element that was modified farther in the past. Since a model fragment is composed by a set of model elements, the timespan weighting of the model fragment depends on the timespan weightings of the model elements that compose it.

The time difference is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. To normalize the time difference, mathematical solutions such as square root or logarithm can be used. We used square root because it has achieved good results in other works that use time differences [62, 27]. The use of square root is more suitable and more effective for the proposed approach [66].

We consider the most recent model modification timespan as the modification timespan weighting of a model fragment. We chose this assessment because it achieved the best results in our previous work [6]. This function expresses the concern of capturing primarily the model fragments with the model elements that have the lowest modification timespans, i.e., model elements that have been recently modified. Then, the value of the model fragment will be the value of the most recently modified model element.

In this work, the modification timespan can come from the metamodel or from the model. If we follow the function presented in [6], in the example of Fig. 9, the value of the model fragment is 7 days, which means a square root of 2.646. However, we can obtain two fitness values, one from the model and another from the metamodel. The value of the model fragment will remain 7 days (a square root of 2.646), while the value of the metamodel fragment will be 20 days (a square root of 4.472). This way, we will obtain two fitness values instead of one. A variant of our BLiM2 applies these two fitness values.

3.3 BLiM2 Variants

When considering the possible variants of the BLiM2 approach, we have taken into account the following facts, which are extracted from the previous subsections:

1. Information retrieval techniques are widely used in bug localization. Specifically LSI, which provides good results applied to source code [50, 35, 46].
2. The Defect Localization Principle improves the results of the LSI when applied together [62, 55, 6].

Fig. 10 shows the variants of BLiM2 taking into account these two facts. The figure shows the BLiM2 approach with the fitness function and the combinations of the information from the model and the metamodel, for a total of ten variants.

1. BLiM2 with one objective in the fitness function taking information from the metamodel in timespan (BLiM2-1OT-MM).
2. BLiM2 with one objective in the fitness function taking information from the model in timespan (BLiM2-1OT-M).
3. BLiM2 with one objective in the fitness function taking information from the metamodel in similarity (BLiM2-1OS-MM).
4. BLiM2 with one objective in the fitness function getting two fitness values from information from the model and the metamodel in similarity and in timespan but combining these objectives in one with equal weight (BLiM2-1O).
5. BLiM2 with two objectives in the fitness function taking information from the model in similarity and timespan (BLiM2-2O-M).
6. BLiM2 with two objectives in the fitness function combining information from the model and the metamodel in similarity and timespan (BLiM2-2O).
7. BLiM2 with two objectives in the fitness function combining information from the model and the metamodel in timespan (BLiM2-2OT).
8. BLiM2 with three objectives in the fitness function combining information from the model and the metamodel in similarity and getting one value for each one in timespan (BLiM2-3OT).
9. BLiM2 with three objectives in the fitness function combining information from the model and the metamodel in timespan and getting one value for each one in similarity (BLiM2-3OS).

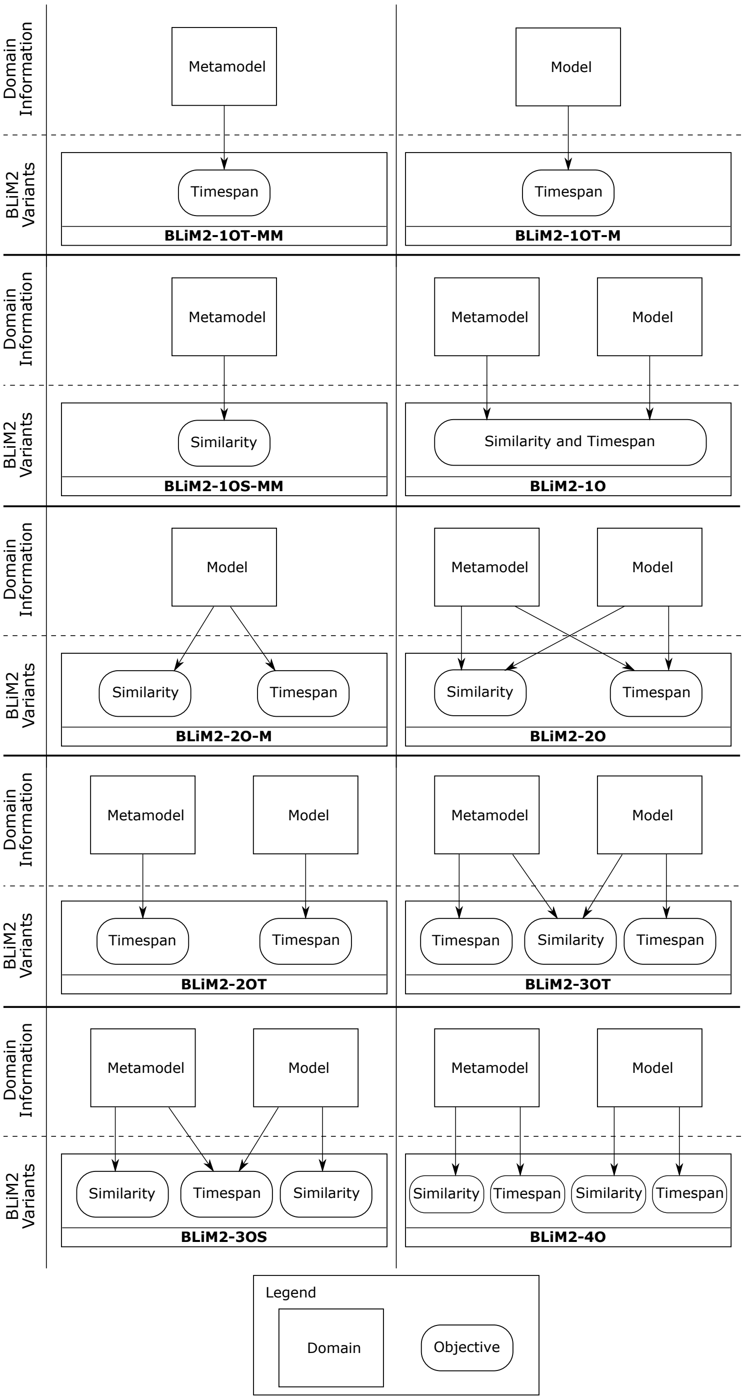


Fig. 10 Variants of the BLiM2 approach

10. BLiM2 with four objectives in the fitness function getting two fitness values from information from the model and the metamodel in similarity and in timespan (BLiM2-4O).

3.4 Implementation Details

Our algorithm is based on NSGA-II [13], one of the most frequently used Multi-Objective Evolutionary Algorithms. Given a set of model fragments where each model fragment has a fitness value for its bug similarity (see section 3.2.1) and its recent time modifications (see section 3.2.2), NSGA-II orders these model fragments by means of non-dominated sorting. A model fragment is non-dominated if the following holds: 1) there is no other model fragment that is better than the current one for some fitness value; and 2) the current model fragment does not worsen other fitness values. As a result, NSGA-II finds pareto-optimal model fragments.

Algorithm 1 BLiM2 (NSGA-II-based)

```
Require: $Size, p_c, p_m, MaxEval$
Ensure: $PF \leftarrow$ set of nondominated solutions
 1: $P = 0$
 2: $evaluations = 0$
 3: for ($i = 1$ to $Size$) do
 4:     $s \leftarrow NewSolution()$
 5:     $EvaluateFitness(s)$
 6:     $evaluations \leftarrow evaluations + 1$
 7:     $P \leftarrow P + 1$
 8: end for
 9: while ($evaluations < MaxEval$) do
10:     $P_O \leftarrow 0$
11:     for ($i = 1$ to $Size/2$) do
12:         $parents \leftarrow Selection(P)$
13:         $offspring \leftarrow Crossover(offspring)$
14:         $offspring \leftarrow Mutation(parents)$
15:         $EvaluateFitness(offspring)$
16:         $evaluations \leftarrow evaluations + 1$
17:         $P_O \leftarrow P_O + offspring$
18:     end for
19:     $P \leftarrow P \cup P_O$
20: end while
21: $PF \leftarrow BestFrom(P)$
22: $RankingAndCrowdingDistance(P)$
```

We implemented the multi-objective evolutionary algorithm as outlined in Algorithm 1. During the generation of the initial set (lines 3-8), model fragments are randomly generated. The algorithm evaluates the fitness for each model fragment and adds it to the initial set. During the evolution process (lines 11-18), new offspring are generated as a result of selecting model fragments (by means of the binary tournament selection), recombining them and applying a mutation operation. The algorithm evaluates the fitness for each offspring and adds it to a temporal set (line 17). By means of a combination of non-dominated sorting and

Table 1 BLiM2 Configuration Parameters

Parameter description	Value
$Size$: population size	100
μ: number of parents	2
λ: number of offspring from μ parents	2
r: solutions replaced by set size	2
p_c: crossover probability	0.9
p_m: mutation probability	0.1

crowding distance sorting [13], the algorithm selects the model fragments from both the old set and the temporal set (line 19) to create a new set.

We performed some parameter tuning to find the best values for the parameters of our algorithm. However, we did not find large differences that had an effect in our main focus. The focus of this paper is not to tune the values to improve the performance of the algorithms when applied to a particular problem, but rather to compare the performance of the algorithms in terms of solution quality. Then, we have principally chosen values for those settings that are commonly used in the literature [54]. As suggested by Arcuri and Fraser [8], default values are good enough to measure the performance of search-based techniques in the context of testing. Hence, the crossover operation is applied with a crossover probability (p_c) of 0.9, and the mutation operation is applied with a probability (p_m) of 0.1.

The number of generations (repetitions of the genetic operations and fitness loop) allowed for the algorithm is 2500 since it is the value needed by our case study to converge (note that this value is case specific). The rest of the settings are detailed in Table 1. There are two atomic performance measures for evolutionary algorithms: one regarding solution quality and another regarding algorithm speed. In this work, we focus on the solution quality, determining the variant that provides solutions that are closer to the one from the oracle in terms of recall, precision, and MCC. Nevertheless, the time spent by each variant to reach the limit of 2500 generations is around 60 seconds.

We have used the Eclipse Modeling Framework to manipulate the models and CVL to manage the fragments of models. The IR techniques used to process the language have been implemented using OpenNLP [2] for the POSTagger and the English version of the Porter 2 [1] as the stemming algorithm. The LSI has been implemented using the Efficient Java Matrix Library (EJML [3]). The genetic operations are built upon the Watchmaker Framework for Evolutionary Computation [16].

4 Evaluation

This section presents the evaluation of our approach: the oracle preparation, the experimental setup, a description of the case study where we applied the evaluation, the results obtained, the statistical analysis performed, the discussion of the results, and the threats to validity.

To evaluate the approach, we applied it to an industrial case study from our industrial partner: BSH, a leading manufacturer of home appliances in Europe.

4.1 Oracle Preparation

The oracle is the ground truth and is used to compare the results provided by the BLiM2-X approaches, the baseline, and the random search (RS) that works as sanity check. The baseline is the approach used in our industrial partner for bug localization [19]. As we explained in section 2.1, a bug can be seen as an unwanted functionality, and a feature represents a functionality. For this reason, the feature localization approach presented in Font et. al [19] can be used for bug localization. Even though it was designed with a more general purpose in mind (Feature Localization), it is the best they have for Bug Localization in Models.

To prepare the oracle, our industrial partner provided us with the bug reports that have occurred in the product models. These bug reports contain natural language bug descriptions and the approved model fragments that contain the target bugs.

4.2 Experimental setup

This experiment evaluates whether or not the information found in the metamodel improves the bug localization results. In addition, we compare the BLiM2-X approaches with the baseline [19] and with a random search (RS) sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use metaheuristic search. We use the name BLiM2-X to represent any of the variants of our approach; the letter 'X' represents one of '1OT-MM', '1OT-M', '1OS-MM', '1O', '2O-M', '2O', '2OT', '3OT', '3OS', and '4O'.

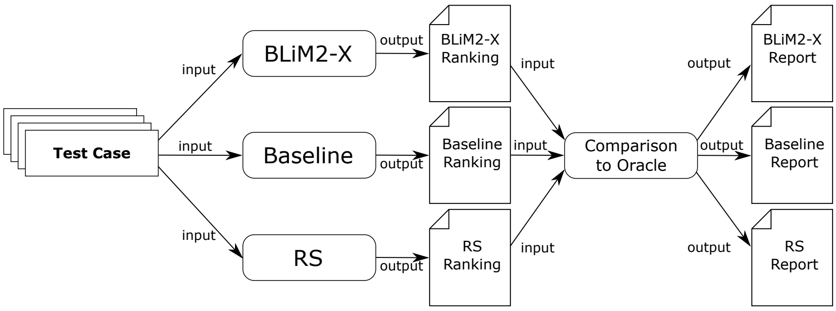
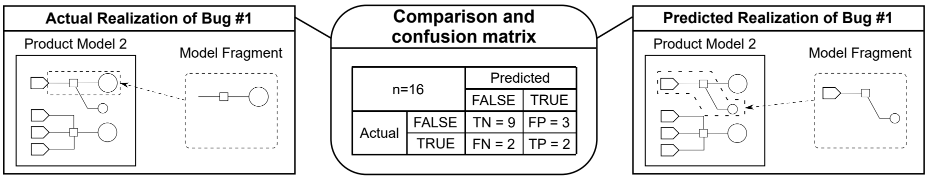


Fig. 11 Evaluation process

Fig. 11 shows an overview of the process that was followed to evaluate our BLiM2-X approach. The left part of the figure shows the inputs of the evaluation process provided by our industrial partner, which are the product family and bug reports. The product family and bug descriptions are used to run BLiM2-X and the baseline approaches. We run each of the approaches and obtain a ranking of model fragments that we can compare with an oracle in order to check accuracy. The inputs of the evaluation process, which are the product family and bug reports, were provided by our industrial partner.


Actual Realization of Bug #1
Product Model 2
Model Fragment
Comparison and confusion matrix
Predicted Realization of Bug #1
Product Model 2
Model Fragment

n=16		Predicted	
		FALSE	TRUE
Actual	FALSE	TN = 9	FP = 3
	TRUE	FN = 2	TP = 2

Fig. 12 Example of the comparison process and the confusion matrix

Therefore, in order to compare the baseline and the RS approaches with BLiM2-X, we first take the best solution of the baseline and RS approaches. Second, we take the best solution of each BLiM2-X. Finally, these solutions are then compared to the model fragment that contains the target bug of the oracle in order to get a confusion matrix.

A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, BLiM2-X and the baseline) on a set of test data (the solutions) for which the true values are known (from the oracles). In our case, each solution that is output by the approaches is a model fragment composed of a subset of the model elements that are present in the product model (where the bug is being located). Since the granularity will be at the level of model elements, the presence or absence of each model element will be considered as a classification. Therefore, our confusion matrices will distinguish between two values (TRUE or presence and FALSE or absence).

Fig. 12 shows an example of the comparison process that is performed to compare a result from one of the evaluated approaches with the ground truth from the oracle and the resulting confusion matrix. We obtain a confusion matrix for each of the solutions predicted by each of the approaches. The left part of Fig. 12 shows the actual model fragment that contains the bug (obtained from the oracle and considered the ground truth) while the right part of Fig. 12 shows the predicted model fragment output by the approach. The confusion matrix arranges the results of the comparison into four categories:

- True positive (TP): A model element present in the predicted model fragment that is also present in the actual model fragment (e.g., the upper power manager model element is a TP).
- True Negative (TN): A model element not present in the predicted model fragment that is not present in the actual model fragment (e.g., the bottom inverter model element is a TN).
- False Positive (FP): A model element present in the predicted model fragment that is not present in the actual model fragment (e.g., the upper inverter model element is a FP).
- False Negative (FN): A model element not present in the predicted model fragment that is present in the actual model fragment (e.g., the upper inductor model element is a FN).

The confusion matrix holds the results of the comparison between the predicted results and the actual results. The result of the sum of all the categories (TP+TN+FP+FN) is the number of model elements (n) of the model that contains the predicted model fragment. However, in order to evaluate the performance of the approach, it is necessary to extract some measurements from the confusion matrix. Then, some performance measurements are derived from the values in the confusion matrix. Specifically, we create a report that includes four performance measurements (recall, precision, the F-measure, and MCC) for each of the test cases for BLiM2-X and the baseline.

Recall measures the number of elements of the solution that are correctly retrieved by the proposed solution and is defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

Precision measures the number of elements from the solution that are correct according to the ground truth (the oracle) and is defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

The F-measure corresponds to the harmonic mean of precision and recall and is defined as follows:

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

Recall values can range between 0% (i.e., no single model element from the model fragment that contains the bug obtained from the oracle is present in any of the model fragments of the solution) to 100% (i.e., all the model elements from the oracle are present in the solution).

Precision values can range between 0% (i.e., no single model fragment from the solution is present in the model fragment that contains the bug obtained from the oracle) to 100% (i.e., all the model fragments from the solution are present in the model fragment that contains the bug from the oracle). A value of 100% precision and 100% recall implies that both the solution and the model fragment that contains the bug from the oracle are the same.

However, none of these measures correctly handle negative examples (TN). The MCC is a correlation coefficient between the observed and predicted binary classifications that takes into account all of the observed values (TP, TN, FP, FN). MCC is a balanced measure which can be used even if the search space and the predicted solution are of very different sizes [11]. For this reason, MCC is one of the best measures for describing a confusion matrix [47]. It is defined as follows:

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

4.3 Case Study

To evaluate the approach, we applied it to an industrial case study from our industrial partner.

4.3.1 BSH

The case study where we applied our evaluation process is the Induction Hob Product Family of our industrial partner (already presented in section 2 as the running example). The oracle is composed of 46 induction hob models, which, on average, are composed of more than 500 elements. Our industrial partner provided us with documentation of 37 bug reports and the approved model fragments that contain the bugs.

The approved model fragments have between 3 and 15 model elements, with an average of 8 model elements. It is important to highlight that each model element has properties (that include terms), and a modification timespan, which are used to differentiate among model elements. Five domain engineers from our industrial partner were involved in providing the set of 37 bugs. The domain engineers of BSH based their selection on a combination of importance and frequency. The set of bugs provided are the most representatives of the bugs that occurs in BSH.

For each of the 37 bugs, we created a test case that includes the set of product models where that bug was manifested and a bug description, both obtained from the documentation. The following video illustrates the models and model fragments of BSH: `youtube.com/watch?v=nS2sybEv6j0`

Each time we run each of the approaches we get one results for a bug. As the approaches performs genetic operations, chance could affect the results. In order to minimize the effect of chance, we execute each of the approaches 30 times for each of the bugs as suggested in [8]. Then, for this case study, we executed 30 independent runs for each of the 37 test cases for BLiM2 and the baseline approach, i.e., 37 (bugs) x 10 (approaches) x 30 repetitions = 11,100 independent runs.

4.4 Results

In this section, we present the results obtained for the case study in BLiM2-X, the baseline, and RS approaches in BSH. Fig. 13 shows the charts with the recall and precision results for the industrial domain. A dot in the graph represents the average result of precision and recall for each of the 37 bugs in BSH for the 30 repetitions.

Table 2 shows the mean values of recall, precision, the F-measure, and MCC for BLiM2-X, the baseline and the random search in the case study. There are five algorithms that obtained best results than the baseline. The BLiM2-3OT approach obtains the best results in recall, precision, and MCC, providing an average value of 89.84% in recall, 80.87% in precision, and 0.82 in MCC. The second best results are obtained by BLiM2-4O, providing an average value of 83.03% in recall, 75.05% in precision, and 0.76 in MCC. The third best values are obtained by BLiM2-2O, providing an average value of 80.76% in recall, 72.56% in precision, and 0.72 in MCC. The fourth best values are obtained by BLiM2-3OS, providing an average value of 73.72% in recall, 67.97% in precision, and 0.66 in MCC. The fifth best values are obtained by BLiM2-1O, providing an average value of 66.26% in recall, 61.29% in precision, and 0.59 in MCC. The rest of the algorithms obtained lower results. In terms of recall, precision, and MCC, BLiM2-3OT outperforms the rest of the approaches.

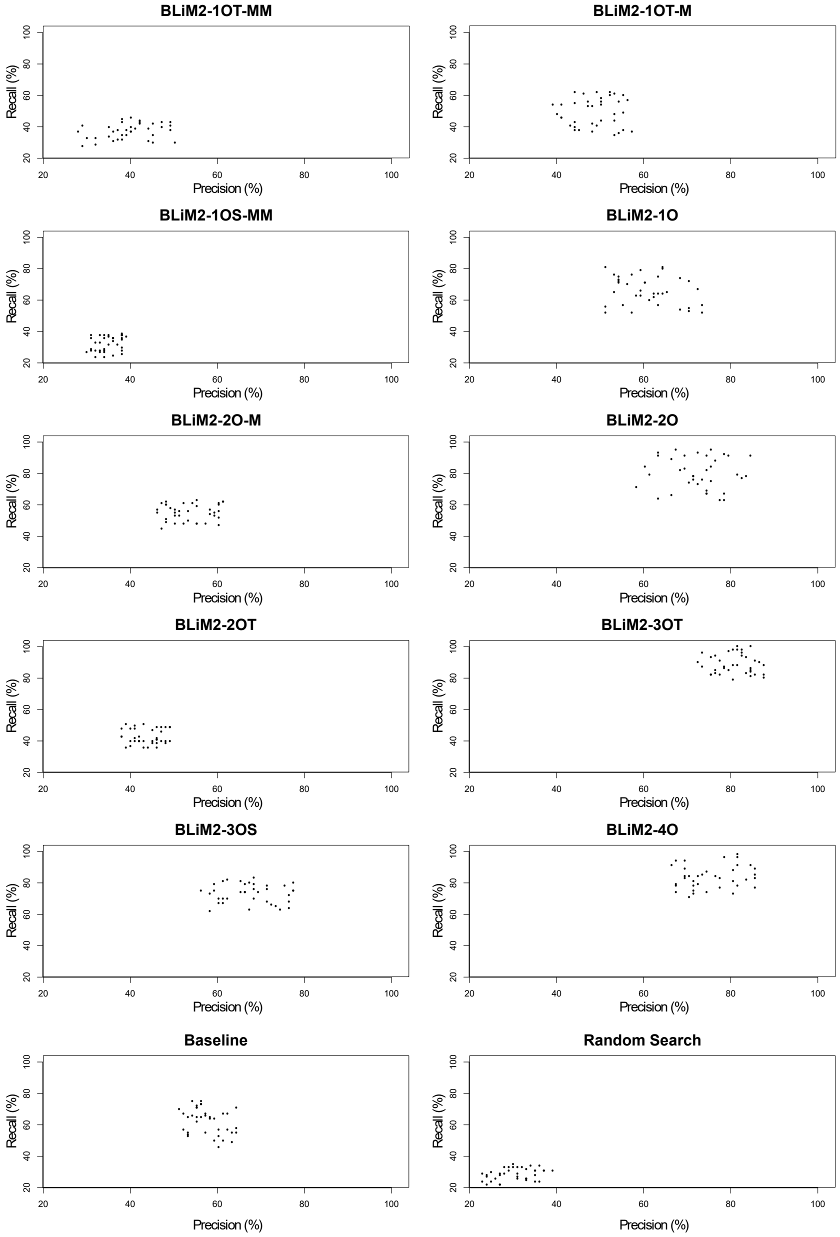


Fig. 13 Mean Recall and Precision values for BLiM2 and baseline approaches in BSH

4.5 Statistical Analysis

To properly compare our BLiM2 approaches and the baseline approach, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [7]. The goals of our statistical analysis are: (1) to

Table 2 Mean values and standard deviations for Recall, Precision, the F-measure, and MCC in BSH

	Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)	MCC ± (σ)
BLiM2-1OT-MM	37.98 ± 5.35	40.66 ± 6.61	37.20 ± 5.60	0.31 ± 0.05
BLiM2-1OT-M	49.11 ± 9.38	48.07 ± 5.40	48.53 ± 5.76	0.43 ± 0.06
BLiM2-1OS-MM	32.03 ± 5.57	35.91 ± 3.91	33.99 ± 3.19	0.25 ± 0.04
BLiM2-1O	66.26 ± 9.70	61.29 ± 7.11	63.43 ± 5.03	0.59 ± 0.05
BLiM2-2O-M	55.17 ± 5.41	53.02 ± 5.33	54.05 ± 4.69	0.48 ± 0.04
BLiM2-2O	80.76 ± 10.95	72.53 ± 7.68	75.34 ± 6.75	0.72 ± 0.07
BLiM2-2OT	43.06 ± 5.64	44.62 ± 3.24	43.82 ± 3.91	0.36 ± 0.04
BLiM2-3OT	89.84 ± 6.01	80.87 ± 4.51	84.18 ± 4.95	0.82 ± 0.04
BLiM2-3OS	73.72 ± 6.93	67.97 ± 6.91	70.39 ± 4.89	0.66 ± 0.05
BLiM2-4O	83.03 ± 7.43	75.05 ± 6.16	79.58 ± 5.66	0.76 ± 0.06
Baseline	62.10 ± 8.42	57.57 ± 4.89	59.49 ± 4.31	0.54 ± 0.04
Random Search	29.05 ± 4.90	30.20 ± 4.15	29.75 ± 3.77	0.21 ± 0.04

provide formal and quantitative evidence (statistical significance) that BLiM2 does in fact have an impact on the comparison metrics (i.e., that the differences in the results were not obtained by mere chance); and (2) to show that those differences are significant in practice (effect size).

4.5.1 Statistical Significance

To enable statistical analysis, all of the algorithms should be run a large enough number of times (in an independent way) to collect information on the probability distribution for each algorithm. A statistical test should then be run to assess whether there is enough empirical evidence to claim (with a high level of confidence) that there is a difference between the two algorithms (e.g., A is better than B). In order to do this, two hypotheses, the null hypothesis H_0 and the alternative hypothesis H_1, are defined. The null hypothesis H_0 is typically defined to state that there is no difference among the algorithms, whereas the alternative hypothesis H_1 states that at least one algorithm differs from another. In such a case, a statistical test aims to verify whether the null hypothesis H_0 should be rejected.

The statistical tests provide a probability value, $p - Value$. The $p - Value$ obtains values between 0 and 1. The lower the $p - Value$ of a test, the more likely that the null hypothesis is false. It is accepted by the research community that a $p - Value$ under 0.05 is statistically significant [8], so the hypothesis H_0 can be considered false.

The test that we must follow depends on the properties of the data. Since our data does not follow a normal distribution in general, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test shows that it is more powerful than the others when working with real data [20]. In addition, according to Conover [12], the Quade test has shown better results than the others when the number of algorithms is low (no more than 4 or 5 algorithms).

The $p - values$ obtained in the test is $\ll 2.2x10^{-16}$ for recall, precision, and MCC; the statistics values obtained are 82.581, 82.412, and 93.53 for recall, precision, and MCC, respectively. Since the $p - values$ are smaller than 0.05 for recall, precision, and MCC, we reject the null hypothesis. Consequently, we can state

Table 3 Holm's post hoc $p - Values$ and the $\hat{A}_{12}$ statistic for each pair of algorithms

	Holm's			$\hat{A}_{12}$		
	Recall	Precision	MCC	Recall	Precision	MCC
3OT vs. 1OT-MM	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1OT-M	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1OS-MM	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 1O	$5.8x10^{-10}$	$1.6x10^{-8}$	$1.4x10^{-11}$	0.99489	0.99672	1
3OT vs. 2O-M	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 2O	0.21685	0.00743	0.00940	0.75347	0.85793	0.89883
3OT vs. 2OT	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. 3OS	$7.1x10^{-5}$	0.00066	$7.0x10^{-5}$	0.98247	0.95435	0.99927
3OT vs. 4O	0.68994	0.57992	0.45910	0.70380	0.73887	0.79657
3OT vs. Baseline	$4.8x10^{-12}$	$1.6x10^{-12}$	$\ll 2.2x10^{-16}$	1	1	1
3OT vs. RS	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	1	1	1

that there are differences among the algorithms for the performance indicators of recall, precision, and MCC.

However, with the Quade test, we cannot answer the following question: Which of the algorithms gives the best performance? In this case, the performance of each algorithm should be individually compared against all other alternatives. In order to do this, we perform an additional post hoc analysis. This kind of analysis performs a pair-wise comparison among the results of each algorithm, determining whether statistically significant differences exist among the results of a specific pair of algorithms.

Table 3 shows the $p - Values$ of Holm's post hoc analysis for the case study and the performance indicators for the algorithm that obtains the best results, BLiM2-3OT [1]. The majority of the $p - Values$ obtained are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between the algorithms are significant. However, when we compare BLiM2-3OT with BLiM2-2O (sixth row), and BLiM2-3OT with BLiM2-4O, the values are greater than the threshold. This indicates that the differences between those algorithms could be due to the stochastic nature of the algorithms and are not significant.

4.5.2 Effect size

When comparing algorithms with a large enough number of runs, statistically significant differences can be obtained even if they are so small as to be of no practical value [8]. Thus it is important to assess if an algorithm is statistically better than another and to assess the magnitude of the improvement. Effect size measures are needed to analyze this.

For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$ [58,22]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than running another algorithm. If the two algorithms are equivalent, then $\hat{A}_{12}$ will be 0.5.

For example, $\hat{A}_{12} = 0.7$ means that we would obtain better results in 70% of the runs with the first of the pair of algorithms that have been compared, and

[1] Although we have performed the entire statistical significance analysis, here we decide to show only the combinations with the algorithm that obtained the best results. Table 5 shows the entire table with the sixty-six combinations (at the end of the paper)

$\hat{A}_{12} = 0.3$ means that we would obtain better results in 70% of the runs with the second of the pair of algorithms that have been compared. Thus, we have an $\hat{A}_{12}$ value for every pair of algorithms.

Table 3 shows the values of the size effect statistics [2]. In general, the largest differences were obtained between BLiM2-3OT and BLiM2-1OT-MM, BLiM2-1OT-M, BLiM2-1OS-MM, BLiM2-2O-M, BLiM2-2OT,the baseline, and the RS, where BLiM2-3OT achieves better results all of the times. When comparing BLiM2-3OT and BLiM2-4O, the differences are not so large, with around 78% in performance.

BLiM2-3OT obtained the best performance results among the twelve evaluated approaches (see Table 2). The performed statistical analysis indicated that BLiM2-3OT outperforms the rest of the approaches in terms of recall, precision, and MCC. Overall, these results confirm that the use of BLiM2-3OT against the baseline approach has an actual impact. In other words, BLiM2-3OT obtained better results in recall, which means that the model fragment proposed as the solution has more relevant elements for the bug that must be located than the model fragments proposed by the baseline approach. In the same way, BLiM2-3OT obtained better results in precision, which means that the model fragment proposed as the solution has less non-relevant elements for the bug that we must be located than the model fragments proposed by the baseline approach.

4.6 Discussion

Our results confirm that the BLiM2 variants and the baseline approaches are better than random search based on the four metrics (recall, precision, F-measure, and MCC) on the BSH case study. Through this study, we concluded that there is empirical evidence to support the significance of the results of our algorithms. Thus, an intelligent algorithm is required to find good solutions to perform bug localization in models.

The BLiM2-3OT variant outperforms the rest of the variants of our approach. However, it did not obtain the perfect solution for a bug in any of the cases. In other words, the model fragments obtained from our approach do not include all of the model elements that contain the bug.

One of the issues that we detected that cause this outcome is the vocabulary mismatch. That means that for a specific concept, the terms used in the bug description are different from the terms used in the models. For example, the bug description includes the word *'current'* to refer to the electrical energy that reaches an inductor. However, in the models, the word *'power'* stands for the same concept. Nevertheless, this issue could be solved by augmenting the Natural language processing (NLP) with a dictionary of synonyms. In the same way, we have also detected cases in which in-house terms are used; for example, instead of using the word *inverter*, the name of a manufacturer is used (*'Fairchild'*). Therefore, the regular dictionary of synonyms would not work in this case. This suggests that the dictionary of synonyms should be refined by domain engineers to include in-home terms.

[2] Although we have performed the entire size effect statistics analysis, here we decide to show only the combinations with the algorithm that obtained the best results. Table 6 shows the entire table with the sixty-six combinations (at the end of the paper)

Another issue is the case in which the bug description is incomplete. For example, in a bug description the following sentence can appear: *'The induction hob crashes when the user selects the power level 9 for a double inductor'*. The engineers understand that the inductors have to reach that power level when the user selects it. However, this sentence also embodies implicit knowledge that is not written but is obvious to the domain engineers: *'The double inductor is formed by two concentric inductors'*, and *'If the pot that is on the top is large enough, both inductors have to reach the power level 9, otherwise, only the central inductor must reach that level.'*. Omitting words in the bug description negatively influences the fitness value of textual similarity since the fitness value of textual similarity is based on the co-occurrence of terms. This suggests that we must make the engineers aware of this issue. They should know that in cases in which the results obtained do not have enough quality and more model elements need to be located, they can reformulate the descriptions of the bugs making the implicit knowledge explicit.

In addition, the variants of our approach that combine model and metamodel terms in one objective obtain better results. This outcome is due to the fact that, in the bug description, terms from the model and the metamodel are mixed. From the metamodel comes general terms that are shared by all the induction hobs, but differentiate the parts. For example, with the words from the metamodel, the approach discriminates between inductors and inverters. From the model come terms that specifies the part of the induction hob. For example, with the words from the model, the approach discriminates between double inductor and single inductor. A complete example is the following sentence from a bug description: *'In the induction hobs that have a triple inductor and a pool inductor, the power booster does not work'*. The terms from the metamodel are: *induction hob, inductor, and power*, and the terms from the model are: *triple, pool, and booster*. Therefore, higher values of textual similarity (co-occurrence of terms) will be reached when comparing the description of the bug with the combined terms of the model and the metamodel.

Therefore, the variants of our approach that combine model and metamodel terms in one objective obtains similar results. However, the differences come from the way in which the Defect Localization Principle is measure. For example, between BLiM2-3OT and BLiM2-2O the difference is around 10% in favor of BLiM2-3OT. In BLiM2-2O, the approach gives one fitness value for the timespan, this produces that there are not so many differences between those that have recently been modified and those that have not.

We believe that the above is especially interesting since it can be an important difference between generic modeling languages and DSLs when locating bugs. Unlike generic modeling languages, such as UML, metamodel elements in DSLs contain domain information. On the one hand, the metaelement *Class* of UML is generic enough to be relevant for different domains. On the other hand, a metaelement such as *Inductor* of the Induction Hobs DSL is relevant for specific domains only. This is also the case of other generic modeling languages, such as BPMN or ARCHIMATE. This suggests that in the hypothetical case that the induction hobs would have been specified with a generic modeling language, the results could have been worse since the terms of the model (e.g., class) would not be similar to the terms of the description of the bug (e.g., inductor). Therefore, we think that specific experiments should be carried out with generic modeling languages to determine the performance with the current approach, assess the need to reformulate the query, or even identify new objectives to guide the bug localization.

Finally, our results confirm the relevance of the Defect Localization Principle to models. The majority of bugs provided by the industrial partner (about 90%) are related to recent modifications. For bugs that are not related to recent modifications, our approach (in spite of including a timespan objective) obtains similar or slightly worse results than the baseline in terms of recall, precision, and MCC. This suggests that, given a bug description where we do not know whether or not the recent modification timespans are relevant, the inclusion of the objective of the Defect Localization Principle, in general, leads to better results than if it is not included. Since modification timespan is important in the localization of bugs, the variants of our approach in which we give more relevance to it obtain better results.

The ten variants of our approach are vulnerable to the following issues: vocabulary mismatch and descriptions with implicit knowledge. For textual similarity, the variants that mix the information from the model and the metamodel (BLiM2-2O and BLiM2-3OT) obtained the best results. For timespan modification, the variants that differentiate between modifications in the model and the metamodel (BLiM2-2OT and BLiM2-4O) obtained the best results. BLiM2-3OT, which combines the above, is the one that obtains the best results in models like those of our industrial partner.

Bugs cannot be located using only textual similarity (as the baseline does) due to the vocabulary mismatch and to the descriptions with implicit knowledge. The reason is that some text is missing or that the text is different between the description of the bug and the models. Bugs can not be located using only the Default Localization Principle either. The reason is that all recent modifications would be suggested as relevant to the bug. Bugs can not be located using only information from the metamodel. The reason is the terms of the metamodel are the same for the specific elements of the model, and for the timespan occurs the same, if a modification is performed in an element of the metamodel the time is the same for all those elements of the models. The combination of model and metamodel information allows the approach to discriminate between the rest of the element and between the specific elements.

Hence, the combination of textual similarity and Defect Localization Principle pays off to locate bugs. Specifically, the combination that gives more weight to the Defect Localization Principle than to the textual similarity. In particular, the best results are obtained by the variant BLiM2-3OT of our approach that has 2 objectives of 3 related to the Defect Localization Principle and 1 of 3 related to textual similarity. It turns out that with 2 of 3 objectives the fragments are prioritized according to the Defect Localization Principle (it happens to about 90% of the bugs) and the third objective (even imperfect due to the vocabulary mismatch and to the descriptions with implicit knowledge) is able to differentiate between the recent modifications not relevant to the bugs and those that are relevant to the bugs.

4.7 Threats to Validity

In this section, we present some of the threats to validity. We follow the guidelines suggested by De Oliveira et. al [44] to identify those that are applicable to this work.

Conclusion validity threats. The first identified threat of this type is *not accounting for random variation.* To address this threat, we considered 30 independent runs for each bug with each algorithm. The second threat is *lack of formal hypothesis and statistical tests.* In this paper, we employed standard statistical analysis following accepted guidelines [8] in order to avoid this threat. The third threat is the *lack of good descriptive analysis.* In this work, we have used precision, recall, the F-measure, and MCC metrics to analyze the confusion matrix obtained from the experiments; however, other metrics could also be applied. In addition, some works argue that the use of the Vargha and Delaney's $\hat{A}_{12}$ metric may be misrepresentative [43] and that the data should be pretransformed before applying it. We did not find any use case for data pretransformation that applies to our case studies.

Internal validity threats. The first identified threat of this type is *poor parameter settings.* In this paper, we used standard values for the algorithms. These values have been tested in similar algorithms for feature localization [36]. In addition, the choice of the k value in the application of SVD can produce suboptimal accuracy when using LSI for software artifacts [45]. However, we plan to evaluate all of the parameters of our algorithm in a future work. The second threat is *lack of clear of data collection tools and procedures.* The set of 37 bugs used in the evaluation has been provided by domain experts of BSH. The set of bugs provided are the most representatives of the bugs that occurs in BSH. The third threat is *lack of real problem instances.* The evaluation of this paper was applied to an industrial case study, BSH.

Construct validity threats. The identified threat is *lack of assessing the validity of cost measures.* To address this threat, we performed a fair comparison between BLiM2-X and the baseline by generating the same number of model fragments and using the same number of fitness evaluations.

External validity threats. The identified threat of this type is *lack of a clear object selection strategy.* This threat is addressed by using an industrial case study. Our instances are collected from real-world problems.

5 Related Work

In recent years, many bug localization approaches have been proposed. These approaches are usually IR-based approaches, and some of them add the Defect Localization principle. Since our Bug Localization in Models approach applies these techniques to models, in this section, we review some relevant works in the literature. Table 4 shows a summary of the main features of these works.

The first block of works in Table 4 shows the works that are related to this paper:

- Kusumoto et al. [29] and Liang et al. [34] apply static program slicing to bug localization. They apply static slicing to reduce the search domain while programmers locate bugs in their programs. In [29], they evaluate this technique

Table 4 Summary of the works cited in the Related Work section

Approach	Information Retrieval	Defect Localization Principle	Locates	Source Code	Model	Metamodel
Kusumoto et al. [29]	No	No	Bugs	Yes	No	No
Liang et al. [34]	No	No	Bugs	Yes	No	No
Mao et al. [39]	No	No	Bugs	Yes	No	No
Alves et al. [4]	No	No	Bugs	Yes	No	No
Gong et al. [21]	No	No	Bugs	Yes	No	No
Saha et al [51]	Yes	No	Bugs	Yes	No	No
Zhou et al. [65]	Yes	No	Bugs	Yes	No	No
Rao et al. [49]	Yes	No	Bugs	Yes	No	No
Lukins et al. [38]	Yes	No	Bugs	Yes	No	No
Kim et al. [28]	Yes	No	Bugs	Yes	No	No
Le et al. [32]	Yes	No	Bugs	Yes	No	No
Hoang et al. [25]	Yes	No	Bugs	Yes	No	No
Lam et al. [30]	Yes	No	Bugs	Yes	No	No
Zamani et al. [62]	Yes	Yes	Bugs	Yes	No	No
Sisman and Kak [55]	Yes	Yes	Bugs	Yes	No	No
Wang and Lo [59]	Yes	Yes	Bugs	Yes	No	No
Wille et al. [60]	No	No	Features	No	Yes	No
Holthusen et al. [26]	No	No	Features	No	Yes	No
Zhang et al. [63,64]	No	No	Features	No	Yes	No
Martinez et al. [42,41]	No	No	Features	No	Yes	No
Lopez-Herrejon et al. [37]	No	No	Features	No	Yes	No
Arcega et al. [5]	Yes	No	Features	No	Yes	No
Font et al. [18,19]	Yes	No	Features	No	Yes	No
Arcega et al. [6]	Yes	Yes	Bugs	No	Yes	No
Our approach	Yes	Yes	Bugs	No	Yes	Yes

and confirm that it is useful for bug localization. In [34], they use this technique to improve the efficiency of data-flow analysis in the presence of pointer variables.

- Mao et al. [39] use dynamic slicing and statistical bug localization. They utilize program slices of a set of test runs to capture the influence of a program entity's execution on the output, and they use statistical analysis to measure the level of suspiciousness of each program entity being faulty. Their approach is called approximate dynamic backward slice.
- In the same way, Alves et al. [4] combine dynamic slicing and spectrum-based techniques. They rank all of the statements in a program based on their level of suspiciousness, which is calculated by using a spectrum-based technique (the Tarantula technique). Then, they generate a dynamic slice with respect to a failure-indicating variable at the failure point. The statements that are not in the slice are removed from the ranking to reduce the search domain.
- Gong et al. [21] propose an interactive localization technique called TALK. This approach incorporates programmers' feedback into spectrum-based fault localization techniques. When the programmer receives the ranking of program elements that can cause the bug, he or she can judge the correctness of each element and provide this information as feedback to re-order the ranking.
- Saha et al. [51], Zhou et al. [65], and Rao et al. [49] apply information retrieval for bug localization. In [51], the authors present BLUiR, Bug Localization Using information Retrieval. In [65], the authors propose BugLocator. Both works use an initial bug report to rank the source code files in descending order based on their relevance to the bug report.
- Lukins et al. [38] used Latent Dirichlet Allocation (LDA) for predicting the location of a newly reported bug. They use source code comments and identifiers as information resources for predicting the locations of bugs.

- Kim et al. [28] propose both a one-phase and a two-phase prediction model to recommend files to fix. In the one-phase model, they create features from textual information and metadata of bug reports, apply Nave Bayes to train the model using previously fixed files as classification labels, and then use the trained model to assign multiple source files to a bug report. In the two-phase model, they first apply their one-phase model to classify a new bug report as either "predictable" or "deficient" and then make predictions only for "predictable" report.
- Le et al. [32] and Hoang et al. [25] combine information retrieval and spectrum-based techniques. In [32,25], they presents two approaches that utilize multi-modal information from both bug reports and program spectra to localize bugs.
- Lam et al. [30] present an approach that uses deep neural network (DNN) in combination with rVSM, an information retrieval technique. rVSM collects the feature on the textual similarity between bug reports and source files. DNN is used to learn to relate the terms in bug reports to potentially different code tokens and terms in source files.
- Zamani et al. [62], Sisman and Kak [55], and Wang and Lo [59] include the Defect Localization principle in their approaches. In [62], they proposed an approach that included weighting and ranking the source code locations based on both the textual similarity with a change request and the use of the time metadata. In [55], they utilize time decay in weighting the files in a probabilistic information retrieval model. In [59], they present AmaLgam+, which is a method for locating relevant buggy files that puts together fives sources of information: version history, similar reports, structure, stack traces, and reporter information.

  These approaches give better results than information retrieval techniques.

All of the above works present approaches that only take into account the source code as the artifact that represents the bug. These approaches have not been applied to models. In contrast, we propose an approach that can be configured to apply these ideas (static and dynamic analysis, information retrieval, and the Defect Localization Principle) in models. We have evaluated the approach successfully by applying them in models. Moreover, our approach does not apply the dynamic analysis that is used by some of those cited. Our future work involves designing an approach that addresses the dynamic analysis idea using models at run-time [9]

Some works focus on the localization of features in models by comparing the models with each other to formalize the variability among them in the form of a Software Product Line:

- Wille et al. [60] present an approach where the similarity between models is measured following an exchangeable metric, taking into account different attributes of the models. Then, the approach is further refined [26] to reduce the number of comparisons needed to mine the family model.
- The authors in [63] propose a generic approach to automatically compare products and locate the feature realizations in terms of a CVL model. In [64], the approach is refined to automatically formalize the feature realizations of new product models that are added to the system.
- Martinez et al. [42] propose an extensible approach that is based on comparisons to extract the feature formalization on a family of models. In addition,

they provide means to extend the approach to locate features in any kind of asset based on comparisons.
- The MoVaPL approach [41] considers the identification of variability and commonality in model variants as well as the extraction of a Model-based Software Product Line (MSPL) from the features identified in these s. MoVaPL builds on a generic representation of models making it suitable for any MOF-based models.
- Lopez-Herrejon et al. [37] evaluate three standard search-based techniques (evolutionary algorithm, hill climbing, and random search) in order to calculate the relationships of a feature model. They calculate the feature relationships of the feature specification layer, while our work locates the model fragments of the product realization

Nevertheless, all of these approaches are based on mechanical comparisons among the models, classifying the elements based on their similarity, and identifying the dissimilar elements as the feature realizations. In contrast, our work does not rely on model comparisons to locate the bugs. Specifically, in our work, we use a multi-objective evolutionary algorithm that uses both the similarity to the bug report and the timespan weighting as fitness functions.

In addition, there are other approaches for feature localization in models that, although they were not designed to locate bugs, could potentially be applied to do this. In the second block of works in Table 4, we show our previous works.

- In [5], we present an approach that is composed of dynamic analysis and information retrieval at the model level. We compare different ways of creating model traces. This work outperforms the feature localization in source code.
- Font et al. [18, 19] propose two approaches that use evolutionary algorithms to locate features in a model. This work does not take into account the Defect Localization Principle for bug localization.
- In [6], we analyze the influence of several timespan weightings on bug localization in models. We evaluate four timespan weightings: the most recent model modifications, the oldest model modifications, the mean of the modification timespan of the modified model elements, and the sum of the modification timespan of the modified model elements. The results show that the use of the most recent timespan model modifications provides the best results in bug localization.

In this work, we adapt the Defect Localization Principle that is used in source code to models. The approach of this paper supports ten different fitness functions, in which the Defect Localization Principle is used. This principle is applied using the most recent timespan model modifications because it obtains the best results in [6]. In addition, in contrast to our previous works, our approach takes into account the domain information from the metamodel.

6 Conclusion

Bug localization is a significant maintenance activity. In this paper, we have proposed an approach for bug localization in models (BLiM2) which enables us to evaluate to what extent the ideas that have been successfully applied in source code for bug localization (textual similarity to bug description and Defect Localization Principle) also work in models. In addition, BLiM2 takes information from two levels, the model and the metamodel, taking advantage of a particularity of the models that does not exist in source code (in the models, there is domain information in both the model and the metamodel).

We evaluated our BLiM2 approach in an industrial case study, BSH, and compared it with the approach that they are using for bug localization. We determined which approach produces the best results in terms of precision, recall, the F-measure, and MCC. To do this, we applied the approaches in BSH that has a model based product family (firmware of Induction Hobs). We present our evaluation, which includes the following: experimental setup, results, statistical analysis, and threats to validity.

The results show that the domain information from the metamodel pays off for bug localization. Specifically, the BLiM2-3OT of our approach achieves the best results. It has three objectives in the fitness function: one that combines information from the model and the metamodel for text similarity; one that takes into account the modification timespan of the model; and one that takes into account the modification timespan of the metamodel. Results also show that our approach can be applied in real world environments. The statistical analysis of the results provides evidence of their significance.

Acknowledgements This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the project Model-Driven Variability Extraction for Software Product Line Adoption (TIN2015-64397-R). We also thank ITEA3 15010 REVaMP2 Project.

References

1. The english (porter2) stemming algorithm. http://snowball.tartarus.org/algorithms /english/stemmer.html (2002). [Online; accessed 04-April-2017]
2. Apache opennlp: Toolkit for the processing of natural language text. http://opennlp.apache.org/ (2010). [Online; accessed 04-April-2017]
3. Efficient java matrix library. https://ejml.org (2016). [Online; accessed 04-April-2017]
4. Alves, E., Gligoric, M., Jagannath, V., d'Amorim, M.: Fault-localization using dynamic slicing and change impact analysis. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pp. 520–523. IEEE Computer Society, Washington, DC, USA (2011). DOI 10.1109/ASE.2011.6100114. URL `http://dx.doi.org/10.1109/ASE.2011.6100114`
5. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: On the influence of models at run-time traces in dynamic feature location. In: Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings (2017)
6. Arcega, L., Font, J., Haugen, Ø., Cetina, C.: On the influence of modification timespan weightings in the location of bugs in models. In: Proceedings of the 26th International Conference on Information Systems Development, ISD 2017, Larnaca, Cyprus, September 6-8, 2017 (2017)
7. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability **24**(3), 219 – 250 (2014). DOI 10.1002/stvr.1486. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486`
8. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering **18**(3), 594–623 (2013). DOI 10.1007/s10664-013-9249-9. URL `http://dx.doi.org/10.1007/s10664-013-9249-9`

9. Bencomo, N., Hallsteinsen, S., Santana de Almeida, E.: A view of the dynamic software product line landscape. Computer **45**(10), 36–41 (2012). DOI 10.1109/MC.2012.292
10. Blei, D.M., Ng, A.Y., Jordan, M.I., Lafferty, J.: Latent dirichlet allocation. Journal of Machine Learning Research **3**(4/5), 993 – 1022 (2003). URL `http://search.ebscohost.com/login.aspx?direct=true&db=bth&AN=12323372&lang=es&site=eds-live`
11. Boughorbel, S., Jarray, F., El-Anbari, M.: Optimal classifier for imbalanced data using matthews correlation coefficient metric. PLOS ONE **12**(6), 1–17 (2017). DOI 10.1371/journal.pone.0177678. URL `https://doi.org/10.1371/journal.pone.0177678`
12. Conover, W.J.: Practical Nonparametric Statistics, 3rd Edition. Wiley (1999)
13. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. Trans. Evol. Comp **6**(2), 182–197 (2002). DOI 10.1109/4235.996017. URL `http://dx.doi.org/10.1109/4235.996017`
14. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science **41**(6), 391–407 (1990). DOI 10.1002/(SICI)1097-4571(199009)41:6⟨391::AID-ASI1⟩3.0.CO;2-9
15. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: A taxonomy and survey. In: Journal of Software Maintenance and Evolution: Research and Practice (2011)
16. Dyer, D.W.: The watchmaker framework for evolutionary computation (evolutionary/genetic algorithms for java). http://watchmaker.uncommons.org/ (2006). [Online; accessed 04-April-2017]
17. Font, J., Arcega, L., ystein Haugen, Cetina, C.: Leveraging variability modeling to address metamodel revisions in model-based software product lines. Computer Languages, Systems & Structures **48**, 20 – 38 (2017). DOI https://doi.org/10.1016/j.cl.2016.08.003. URL `http://www.sciencedirect.com/science/article/pii/S147784241630001X`. Special Issue on the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE'15)
18. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in model-based software product lines through a genetic algorithm. In: 15th International Conference on Software Reuse, ICSR 2016. Limassol, Cyprus (2016)
19. Font, J., Arcega, L., Haugen, Ø., Cetina, C.: Feature location in models through a genetic algorithm driven by information retrieval techniques. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16, pp. 272–282. ACM, New York, NY, USA (2016). DOI 10.1145/2976767.2976789. URL `http://doi.acm.org/10.1145/2976767.2976789`
20. Garca, S., Fernndez, A., Luengo, J., Herrera, F.: Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. Information Sciences **180**(10), 2044 – 2064 (2010). DOI http://dx.doi.org/10.1016/j.ins.2009.12.010. URL `http://www.sciencedirect.com/science/article/pii/S0020025509005404`. Special Issue on Intelligent Distributed Information Systems
21. Gong, L., Lo, D., Jiang, L., Zhang, H.: Interactive fault localization leveraging simple user feedback. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 67–76 (2012). DOI 10.1109/ICSM.2012.6405255
22. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach. Mahwah, NJ: Earlbaum (2005)
23. Hassan, A.E., Holt, R.C.: The top ten list: dynamic fault prediction. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 263–272 (2005). DOI 10.1109/ICSM.2005.91
24. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference, SPLC '08, pp. 139–148. IEEE Computer Society, Washington, DC, USA (2008). DOI 10.1109/SPLC.2008.25
25. Hoang, T.V., Oentaryo, R.J., Le, T.B., Lo, D.: Network-clustered multi-modal bug localization. IEEE Transactions on Software Engineering pp. 1–1 (2018). DOI 10.1109/TSE.2018.2810892
26. Holthusen, S., Wille, D., Legat, C., Beddig, S., Schaefer, I., Vogel-Heuser, B.: Family model mining for function block diagrams in automation software. In: Proceedings of the 18th International Software Product Line Conference: Volume 2, pp. 36–43 (2014). DOI 10.1145/2647908.2655965

27. Kagdi, H., Gethers, M., Poshyvanyk, D., Hammad, M.: Assigning change requests to software developers. Journal of Software: Evolution and Process **24**(1), 3–33 (2012). DOI 10.1002/smr.530. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.530`
28. Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? a two-phase recommendation model. IEEE Transactions on Software Engineering **39**(11), 1597–1610 (2013)
29. Kusumoto, S., Nishimatsu, A., Nishie, K., Inoue, K.: Experimental evaluation of program slicing for fault localization. Empirical Softw. Engg. **7**(1), 49–76 (2002). DOI 10.1023/A:1014823126938. URL `http://dx.doi.org/10.1023/A:1014823126938`
30. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 218–229 (2017). DOI 10.1109/ICPC.2017.24
31. Landauer, T.K., Foltz, P.W., Laham, D.: An introduction to latent semantic analysis. Discourse Processes **25**(2-3), 259–284 (1998). DOI 10.1080/01638539809545028. URL `http://dx.doi.org/10.1080/01638539809545028`
32. Le, T.D.B., Oentaryo, R.J., Lo, D.: Information retrieval and spectrum based bug localization: Better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 579–590. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786880. URL `http://doi.acm.org/10.1145/2786805.2786880`
33. Lehman, M.M., Ramil, J., Kahen, G.: A paradigm for the behavioural modelling of software processes using system dynamics. Tech. rep., Imperial College of Science, Technology and Medicine, Department of Computing (2001)
34. Liang, D., Harrold, M.J.: Equivalence analysis and its application in improving the efficiency of program slicing. ACM Trans. Softw. Eng. Methodol. **11**(3), 347–383 (2002). DOI 10.1145/567793.567796. URL `http://doi.acm.org/10.1145/567793.567796`
35. Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V.: Feature location via information retrieval based filtering of a single scenario execution trace. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 234–243. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321667. URL `http://doi.acm.org/10.1145/1321631.1321667`
36. Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. Journal of Systems and Software **103**, 353 – 369 (2015). DOI http://dx.doi.org/10.1016/j.jss.2014.10.037
37. Lopez-Herrejon, R.E., Linsbauer, L., Galindo, J.A., Parejo, J.A., Benavides, D., Segura, S., Egyed, A.: An assessment of search-based techniques for reverse engineering feature models. Journal of Systems and Software **103**, 353 – 369 (2015). DOI http://dx.doi.org/10.1016/j.jss.2014.10.037. URL `http://www.sciencedirect.com/science/article/pii/S0164121214002349`
38. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent dirichlet allocation. Inf. Softw. Technol. **52**(9), 972–990 (2010). DOI 10.1016/j.infsof.2010.04.002. URL `http://dx.doi.org/10.1016/j.infsof.2010.04.002`
39. Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C.: Slice-based statistical fault localization. J. Syst. Softw. **89**, 51–62 (2014). DOI 10.1016/j.jss.2013.08.031. URL `http://dx.doi.org/10.1016/j.jss.2013.08.031`
40. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.: An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 214–223 (2004). DOI 10.1109/WCRE.2004.10
41. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., l. Traon, Y.: Automating the extraction of model-based software product lines from model variants (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, pp. 396–406 (2015). DOI 10.1109/ASE.2015.44
42. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proceedings of the 19th International Conference on Software Product Line (SPLC), pp. 101–110 (2015). DOI 10.1145/2791060.2791086
43. Neumann, G., Harman, M., Poulding, S.: Transformed Vargha-Delaney Effect Size, pp. 318–324. Springer International Publishing, Cham (2015). URL `http://dx.doi.org/10.1007/978-3-319-22183-0_29`
44. de Oliveira Barros, M., Dias-Neto, A.C.: 0006/2011-threats to validity in search-based software engineering empirical studies. RelaTe-DIA **5**(1) (2011)

45. Panichella, A., Dit, B., Oliveto, R., Penta, M.D., Poshyvanyk, D., Lucia, A.D.: Parameterizing and assembling ir-based solutions for se tasks using genetic algorithms. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 314–325 (2016). DOI 10.1109/SANER.2016.97
46. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Transactions on Software Engineering **33**(6), 420–432 (2007). DOI 10.1109/TSE.2007.1016. URL `http://dx.doi.org/10.1109/TSE.2007.1016`
47. Powers, D.M.W.: Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. Journal of Machine Learning Technologies **2**(1), 37–63 (2011)
48. Rahman, M.M., Chakraborty, S., Ray, B.: Which similarity metric to use for software documents?: A study on information retrieval based software engineering tasks. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, pp. 335–336. ACM, New York, NY, USA (2018). DOI 10.1145/3183440.3194997. URL `http://doi.acm.org/10.1145/3183440.3194997`
49. Rao, S., Kak, A.: Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, pp. 43–52. ACM, New York, NY, USA (2011). DOI 10.1145/1985441.1985451. URL `http://doi.acm.org/10.1145/1985441.1985451`
50. Revelle, M., Dit, B., Poshyvanyk, D.: Using data fusion and web mining to support feature location in software. In: IEEE 18th International Conference on Program Comprehension (ICPC), pp. 14–23 (2010). DOI 10.1109/ICPC.2010.10
51. Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 345–355 (2013). DOI 10.1109/ASE.2013.6693093
52. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA (1986)
53. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11), 613–620 (1975). DOI 10.1145/361219.361220. URL `http://doi.acm.org/10.1145/361219.361220`
54. Sayyad, A.S., Ingram, J., Menzies, T., Ammar, H.: Scalable product line configuration: A straw to break the camel's back. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 465–474 (2013). DOI 10.1109/ASE.2013.6693104
55. Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 50–59 (2012). DOI 10.1109/MSR.2012.6224299
56. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a software product line for train control: A case study of cvl. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, pp. 106–120. Springer-Verlag, Berlin, Heidelberg (2010). URL `http://dl.acm.org/citation.cfm?id=1885639.1885650`
57. Thomas, S.W., Hassan, A.E., Blostein, D.: Mining Unstructured Software Repositories, pp. 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). DOI 10.1007/978-3-642-45398-4_5. URL `https://doi.org/10.1007/978-3-642-45398-4_5`
58. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000). DOI 10.3102/10769986025002101
59. Wang, S., Lo, D.: Amalgam+: Composing rich information sources for accurate bug localization. Journal of Software: Evolution and Process **28**(10), 921–942 (2016). DOI 10.1002/smr.1801. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1801`
60. Wille, D., Holthusen, S., Schulze, S., Schaefer, I.: Interface variability in family model mining. In: Proceedings of the 17th International Software Product Line Conference: Co-located Workshops, pp. 44–51 (2013). DOI 10.1145/2499777.2500708
61. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016)
62. Zamani, S., Lee, S.P., Shokripour, R., Anvik, J.: A noun-based approach to feature location using time-aware term-weighting. Information and Software Technology **56**(8), 991 – 1011 (2014). DOI http://dx.doi.org/10.1016/j.infsof.2014.03.007. URL `http://www.sciencedirect.com/science/article/pii/S0950584914000688`

63. Zhang, X., Haugen, Ø., Moller-Pedersen, B.: Model comparison to synthesize a model-driven software product line. In: Proceedings of the 2011 15th International Software Product Line Conference (SPLC), pp. 90–99 (2011). DOI 10.1109/SPLC.2011.24
64. Zhang, X., Haugen, Ø., Møller-Pedersen, B.: Augmenting product lines. In: Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, vol. 1, pp. 766–771 (2012). DOI 10.1109/APSEC.2012.76
65. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 14–24. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337226
66. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 563–572. IEEE Computer Society, Washington, DC, USA (2004). URL http://dl.acm.org/citation.cfm?id=998675.999460

Table 5 Holm's post hoc $p - Values$ for recall for each pair of algorithms

	Recall										
	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.01218	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.22381	$8.4x10^{-7}$	-	-	-	-	-	-	-	-	-
1O	$1.2x10^{-14}$	$1.1x10^{-5}$	$\ll 2.2x10^{-16}$	-	-	-	-	-	-	-	-
2O-M	$2.1x10^{-6}$	0.27833	$3.5x10^{-12}$	0.03263	-	-	-	-	-	-	-
2O	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	0.00013	$5.9x10^{-12}$	-	-	-	-	-	-
2OT	0.65083	0.35431	0.00340	$3.3x10^{-10}$	0.00171	$\ll 2.2x10^{-16}$	-	-	-	-	-
3OT	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$5.8x10^{-10}$	$\ll 2.2x10^{-16}$	0.21685	$\ll 2.2x10^{-16}$	-	-	-	-
3OS	$\ll 2.2x10^{-16}$	$4.6x10^{-11}$	$\ll 2.2x10^{-16}$	0.22577	$5.7x10^{-6}$	0.21685	$\ll 2.2x10^{-16}$	$7.1x10^{-5}$	-	-	-
4O	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$1.4x10^{-6}$	$6.5x10^{-15}$	0.93749	$\ll 2.2x10^{-16}$	0.68994	0.01387	-	-
Baseline	$2.4x10^{-12}$	0.00031	$\ll 2.2x10^{-16}$	0.93749	0.21685	$4.4x10^{-6}$	$2.8x10^{-8}$	$4.8x10^{-12}$	0.04335	$2.3x10^{-8}$	-
RS	0.01795	$2.7x10^{-9}$	0.93749	$\ll 2.2x10^{-16}$	$2.9x10^{-15}$	$\ll 2.2x10^{-16}$	$5.7x10^{-5}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$

	Precision										
	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.00650	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.43504	$2.4x10^{-6}$	-	-	-	-	-	-	-	-	-
1O	$\ll 2.2x10^{-16}$	$4.6x10^{-7}$	$\ll 2.2x10^{-16}$	-	-	-	-	-	-	-	-
2O-M	$4.7x10^{-7}$	0.32239	$6.3x10^{-12}$	0.00650	-	-	-	-	-	-	-
2O	$\ll 2.2x10^{-16}$	$6.0x10^{-15}$	$\ll 2.2x10^{-16}$	0.05708	$1.1x10^{-8}$	-	-	-	-	-	-
2OT	0.43504	0.43504	0.00625	$7.2x10^{-12}$	0.00137	$\ll 2.2x10^{-16}$	-	-	-	-	-
3OT	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$1.6x10^{-8}$	$\ll 2.2x10^{-16}$	0.00743	$\ll 2.2x10^{-16}$	-	-	-	-
3OS	$\ll 2.2x10^{-16}$	$8.4x10^{-13}$	$\ll 2.2x10^{-16}$	0.32239	$4.8x10^{-7}$	0.57992	$\ll 2.2x10^{-16}$	0.00066	-	-	-
4O	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$2.5x10^{-5}$	$1.1x10^{-14}$	0.32239	$\ll 2.2x10^{-16}$	0.57992	0.06481	-	-
Baseline	$1.2x10^{-12}$	0.00045	$\ll 2.2x10^{-16}$	0.57992	0.32239	0.00044	$5.8x10^{-8}$	$1.6x10^{-12}$	0.00616	$1.2x10^{-8}$	-
RS	0.01619	$9.8x10^{-10}$	0.57992	$\ll 2.2x10^{-16}$	$3.1x10^{-16}$	$\ll 2.2x10^{-16}$	$2.0x10^{-5}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$

	MCC										
	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.0076	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.1407	$1.6x10^{-7}$	-	-	-	-	-	-	-	-	-
1O	$\ll 2.2x10^{-16}$	$9.1x10^{-7}$	$\ll 2.2x10^{-16}$	-	-	-	-	-	-	-	-
2O-M	$2.6x10^{-6}$	0.4540	$1.2x10^{-12}$	0.0042	-	-	-	-	-	-	-
2O	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	0.0010	$2.2x10^{-12}$	-	-	-	-	-	-
2OT	0.4591	0.4540	0.0015	$8.4x10^{-12}$	0.0021	$\ll 2.2x10^{-16}$	-	-	-	-	-
3OT	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$1.4x10^{-11}$	$\ll 2.2x10^{-16}$	0.0094	$\ll 2.2x10^{-16}$	-	-	-	-
3OS	$\ll 2.2x10^{-16}$	$3.2x10^{-14}$	$\ll 2.2x10^{-16}$	0.0637	$8.7x10^{-9}$	0.4591	$\ll 2.2x10^{-16}$	$7.0x10^{-5}$	-	-	-
4O	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$3.5x10^{-7}$	$\ll 2.2x10^{-16}$	0.4591	$\ll 2.2x10^{-16}$	0.4591	0.0309	-	-
Baseline	$1.1x10^{-10}$	0.0042	$\ll 2.2x10^{-16}$	0.4540	0.4591	$1.2x10^{-7}$	$7.5x10^{-7}$	$\ll 2.2x10^{-16}$	$7.8x10^{-5}$	$3.2x10^{-12}$	-
RS	0.0047	$1.6x10^{-10}$	0.4591	$\ll 2.2x10^{-16}$	$2.6x10^{-16}$	$\ll 2.2x10^{-16}$	$9.0x10^{-6}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$	$\ll 2.2x10^{-16}$

Table 6 $\hat{A}_{12}$ statistic for each pair of algorithms

	Recall										
↓ vs. →	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.85866	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.24690	0.04456	-	-	-	-	-	-	-	-	-
1O	1	0.89664	1	-	-	-	-	-	-	-	-
2O-M	0.99891	0.68627	1	0.16143	-	-	-	-	-	-	-
2O	1	1	1	0.83638	0.99927	-	-	-	-	-	-
2OT	0.76479	0.29401	0.95435	0	0.05917	0	-	-	-	-	-
3OT	1	1	1	0.99489	1	0.75347	1	-	-	-	-
3OS	1	0.99890	1	0.72498	0.99744	0.30241	1	0.01753	-	-	-
4O	1	1	1	0.93755	1	0.60189	1	0.29620	0.85757	-	-
Baseline	0.99963	0.84697	1	0.39262	0.75566	0.09533	0.98319	0	0.15194	0.01278	-
RS	0.09204	0.00037	0.27904	0	0	0	0	0	0	0	0

	Precision										
↓ vs. →	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.8656	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.2297	0.00037	-	-	-	-	-	-	-	-	-
1O	1	0.93644	1	-	-	-	-	-	-	-	-
2O-M	0.9661	0.73411	1	0.19065	-	-	-	-	-	-	-
2O	1	1	1	0.88093	0.96676	-	-	-	-	-	-
2OT	0.7141	0.23192	0.98722	0	0.04602	0	-	-	-	-	-
3OT	1	1	1	0.99671	1	0.85793	1	-	-	-	-
3OS	1	0.99890	1	0.74981	0.95617	0.30131	1	0.04565	-	-	-
4O	1	1	1	0.93353	1	0.60738	1	0.26114	0.80022	-	-
Baseline	1	0.91965	1	0.35245	0.72899	0.03324	1	0	0.09788	0	-
RS	0.10811	0.00037	0.21110	0	0	0	0.00292	0	0	0	0

	MCC										
↓ vs. →	1OT-MM	1OT-M	1OS-MM	1O	2O-M	2O	2OT	3OT	3OS	4O	Baseline
1OT-M	0.91563	-	-	-	-	-	-	-	-	-	-
1OS-MM	0.20088	0.00292	-	-	-	-	-	-	-	-	-
1O	1	0.97918	1	-	-	-	-	-	-	-	-
2O-M	0.99598	0.77283	1	0.05696	-	-	-	-	-	-	-
2O	1	1	1	0.94668	1	-	-	-	-	-	-
2OT	0.77356	0.19722	0.98685	0	0.01534	0	-	-	-	-	-
3OT	1	1	1	1	1	0.89883	1	-	-	-	-
3OS	1	1	1	0.85646	0.99708	0.24507	1	0.00073	-	-	-
4O	1	1	1	0.99270	1	0.66983	1	0.20343	0.91271	-	-
Baseline	1	0.94595	1	0.24653	0.84222	0.01169	1	0	0.03287	0	-
RS	0.07779	0	0.24836	0	0	0	0	0	0	0	0