Bug Localization in Model-based Systems in the Wild

LORENA ARCEGA, Universidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain
JAIME FONT ARCEGA, Universidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain
ØYSTEIN HAUGEN, Østfold University College. Faculty of Computer Science. Halden, Norway
CARLOS CETINA, Universidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain

The companies that have adopted the Model-Driven Engineering (MDE) paradigm have the advantage of working at a high level of abstraction. Nevertheless, they have the disadvantage of the lack of tools available to perform bug localization at the model level. In addition, in an MDE context, a bug can be related to different MDE artefacts, such as design-time models, model transformations, or run-time models. Starting the bug localization in the wrong place or with the wrong tool can lead to a result that is unsatisfactory. We evaluate how to apply the existing model-based approaches in order to mitigate the effect of starting the localization in the wrong place. We also take into account that software engineers can refine the results at different stages. In our evaluation, we compare different combinations of the application of bug localization approaches and human refinement. The combination of our approaches plus manual refinement obtains the best results. We performed a statistical analysis to provide evidence of the significance of the results. The conclusions obtained from this evaluation are: humans have to be involved at the right time in the process (or results can even get worse), and artefact-independence can be achieved without worsening the results.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; **Software maintenance tools**; • **Information systems → Information retrieval**.

Additional Key Words and Phrases: Bug Localization, Models at runtime, Model-Driven Engineering

ACM Reference Format:
Lorena Arcega, Jaime Font Arcega, Øystein Haugen, and Carlos Cetina. 2020. Bug Localization in Model-based Systems in the Wild. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2020), 33 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Model-Driven Engineering (MDE) is being applied in an ever-increasing manner to cope with the complexity of software systems by raising the level of abstraction [35] in almost every area of software systems. Models, such as design-time models, are artefacts with key roles during the software development process and have successfully been used in many domains. Run-time models are envisioned to provide intelligent support to software during execution [8]. The run-time models are defined as *a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective* [9].

The companies that have adopted the MDE paradigm have the advantage of working at a high level of abstraction (compared to working at the source code level). Nevertheless, a recent survey

Authors' addresses: Lorena ArcegaUniversidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain, larcega@usj.es; Jaime Font ArcegaUniversidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain, jfont@usj.es; Øystein HaugenØstfold University College. Faculty of Computer Science. Halden, Norway, oystein.haugen@hiof.no; Carlos CetinaUniversidad San Jorge. Escuela de Arquitectura y Tecnología. Zaragoza. Spain, ccetina@usj.es.

1049-331X/2020/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

[53] reveals that none of the bug localization approaches take into account models as the source of the bugs. Hence, they have the disadvantage of a lack of approaches that are available to perform bug localization at the model level. In our previous works [2], we have presented several approaches to perform bug localization: in systems with design-time models [4] and systems with run-time models and reconfiguration rules [3].

In addition, in an MDE context, a bug can be related to different MDE elements, such as design-time models, model transformations, or run-time models. Nowadays, each approach for bug localization is focused on an MDE element. For example, BLiMEA [4] works with design-time models and EBRo [3] works with run-time models. Hence, in a real environment, there are times that the information provided about the bug is not enough or is too confusing to be able to decide where to start the localization [30].

Starting the bug localization in the wrong place or with the wrong tool can lead to a result that is unsatisfactory. For example, if the bug is caused by the reconfigurations and the localization is performed with a tool that only takes into account design-time models, the task becomes almost impossible (and the same with the opposite situation). This makes the task of bug localization one of the most important, expensive, tedious, and time-consuming activities in program maintenance.

In this paper, we evaluate how to apply the existing model-based approaches in order to mitigate the effect of starting the localization in the wrong place. In our evaluation, we compare our approaches (BLiMEA and EBRo) with other approaches for bug localization: a coevolutionary algorithm based on BLiMEA and EBRo (CoEB), a hill-climbing algorithm (HC), and a combination of the application of our approaches (EB). In addition, we take into account the fact that software engineers can refine the results. Hence, the action of humans is taken into account in these approaches. Furthermore, we compare our approach with a random search approach (RS) as a sanity check. The evaluation is performed with two real datasets from our partners that contain bugs in different MDE elements: design-time models and run-time models.

The results show that the combination of the EBRo and BLiMEA approaches plus manual refinement outperforms the rest of the approaches. The results in terms of recall, precision, and Matthews Correlation Coefficient (MCC), on average, are 0.90, 0.80, and 0.82, respectively. The statistical analysis performed provides quantitative evidence of the impact of the approaches and indicates that this impact is significant.

Based on our experiences from the evaluations performed, we realized that involving humans must be done at the right time. If this is not done at the right time, the results can even get worse. We must stop considering involving humans as something binary (good or bad). Humans have different types of knowledge that they can contribute to algorithms. Therefore, we should evaluate when and what knowledge is good for each problem.

Finally, we realized that, by combining the EBRo and BLiMEA approaches, the results can be improved somewhat without worsening. In other words, the combination of EBRo and BLiMEA improves the results without running the risk of doing the opposite, namely worsening the process. This combination ensures that the solution is assessed taking into account the run-time and the design-time information. Hence, we can combine our approaches without software engineers having to choose which approach to use to perform the bug localization.

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 motivation. In Section 4, we describe our approaches, BLiMEA and EBRo. In Section 6, we evaluate the combinations of our approach in BSH and discuss the results. In Section 7, we examine the related work of the area. Finally, we present our conclusions in Section 8.

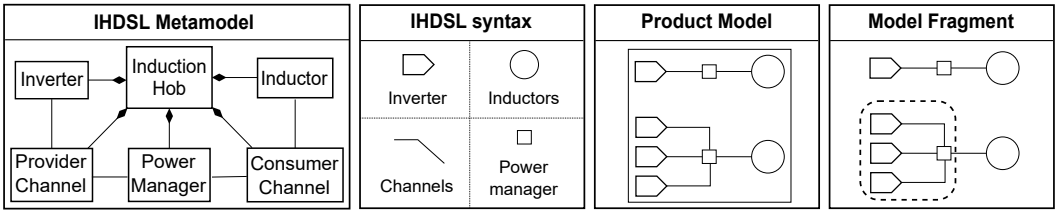

Fig. 1. IHDSL metamodel, syntax, product model, and model fragment

2 BACKGROUND

The running example and the evaluation in this paper are performed using the products of our industrial partner, BSH. BSH is 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.

In this section, we present the Domain-Specific Language (DSL) that is used by BSH to formalize their products. This DSL is used to describe the models at design time and the models at runtime of the induction hobs to which we apply our approach. The firmware of the BSH products is generated from the DSL models. The Common Variability Language (CVL) [26] is also presented. The different configurations of the induction hobs are managed following a model-based Software Product Line (SPL) approach that uses CVL to configure their models.

The newest Induction Hobs (IHs)[1] 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 them. These dynamic areas are managed at runtime by calculating the resulting model after the changes in the context of the IH.

The Domain-Specific Language used by our industrial partner to specify the Induction Hobs (IHDSL) is composed of 46 metaclasses, 47 references among them, and more than 180 properties. For legibility reasons and due to intellectual property rights 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. The Product Model in Figure 1 depicts an example of a model at design time that is specified with the IHDSL.

Inverters are in charge of transforming the 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 runtime. 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.

To formalize the solution of our approach as model fragments, we use Common Variability Language (CVL) [26, 47] because of its capabilities to formalize a set of model elements as a model fragment. The Model Fragment in Fig. 1 shows an example of a model fragment of the product model (the Product Model in Fig. 1). 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 shown to the engineers.

[1]Gaggenau Induction Hob - CX 480: https://www.youtube.com/watch?v=HjZ_nB-TY7w

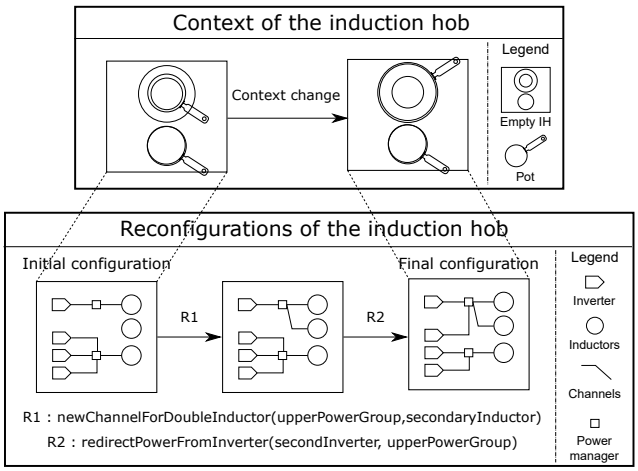

Fig. 2. Induction Hob at runtime

Fig. 2 shows the behavior of an Induction Hob at runtime. In the initial configuration, the Induction Hob has two pots on top that are heated through two inductors. The upper inductor is powered with one inverter and the bottom inductor is powered with three inverters. When a bigger pot is placed on the upper inductor, the Induction Hob reconfigures itself. Another inductor is activated ('R1'), and more energy is needed to heat the pot ('R2'). Therefore, the second inverter is disconnected from the lower power group and is connected to the upper power group. In other words, when changes in the context occur, reconfigurations are triggered in order to change the configuration of the IH. Thus, the Induction Hob is in a different configuration and therefore in a different model. Some other examples of relevant context changes include the following: putting a pot on top of the IH, the pot reaches the set temperature, the pot is moved to another place on the IH, or liquid spills from the pot onto the surface.

3 MOTIVATION

Bug localization is a task that is related to software maintenance and evolution. Removing bugs is analogous to removing unwanted functionality [19]. In the end, software engineers obtain a piece of software that is in charge of some functionality in the system. We have applied our approaches to several model-based systems. Fig. 3 shows the systems and how they are related to each other. This paper focuses on reconfigurable model-based systems.

Reconfigurable model-based systems have some elements that are prone to having bugs. These elements are the metamodel, design-time models, run-time models, and reconfiguration rules. Our previous approaches for bug localization take into account the particularities of bug localization as well as the particularities of the model-based software systems. Table 1 summarizes the artefacts used by each of our previous approaches.

Our BLiMEA approach [4] uses an evolutionary algorithm that iterates through the models of the system and assesses each one of the possible solutions. We use two objectives for assessing each one of the solutions: textual similarity with the bug description combining information from the model and the metamodel; and timespan for measuring the last modification of the model. Hence, the artefacts of the system that are used for the localization by this approach are bug reports, design-time models, and the metamodel (see Table 1, second row).

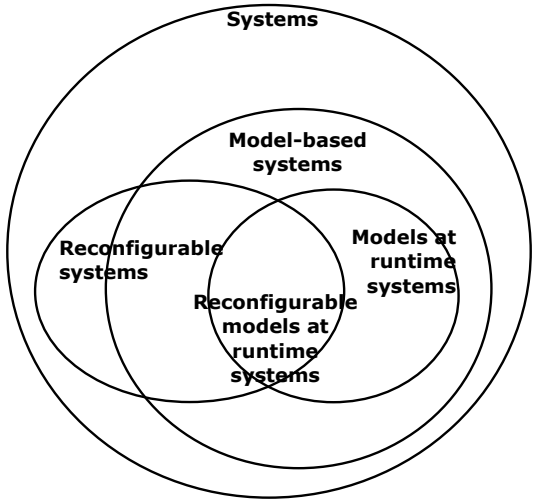

Fig. 3. Systems in which we have applied our approaches

	Bug report	Metamodel	Design time models	Run-time models	Reconfiguration rules
BLiMEA	✓	✓	✓	×	×
EBRo	✓	×	✓	✓	✓

Table 1. Artefacts used in each approach

Our EBRo approach [3] focuses on locating bugs that appear as the result of dynamic reconfigurations of the system due to context changes. Similar to BLiMEA, we use an evolutionary algorithm to iterate through the models. However, the solutions of this approach are sequences of reconfigurations that, when followed, might lead to the model at runtime that contains the bug. Hence, the artefacts of the system used for the localization by this approach are bug reports, design-time models, run-time models, and reconfiguration rules (see Table 1, third row).

In real industrial environments, like the one used in our evaluations [3, 4], when a bug appears, the software engineers usually do not know the artefact that causes the bug. In addition, we realized that if the localization is performed using the wrong approach, the artefact that caused the bug cannot be reached. Our previous approaches take into account the artefacts that are involved in MDE separately, i.e., each approach uses only some of the artefacts. Therefore, if the bug is produced in the run-time models and the software engineers start the localization with BLiMEA (which only takes into account the design-time models), they may never obtain a good solution for discovering the bug. It is important to distinguish the discovery of bugs from their correction. When we talk about solutions for bugs in this paper, we are referring to discovering them; the correction of bugs is out of the scope of this paper.

Therefore, since software engineers do not have to know the artefact where the bug is located, we want to evaluate the best combination for the real cases where they do not know where the bug is. In the evaluation, we consider the approaches presented above and the possibility that software engineers can manually refine the solutions for discovering the bug.

4 THE BUG LOCALIZATION APPROACHES

This section presents the bug localization approaches that are used in our evaluation: BLiMEA [4], EBRo [3], CoEB [12, 37], and Hill Climbing [20, 34]. For each of them, we describe the following: an overview of the approach, the artefacts used as input, the search strategy used to explore the search space, the form of the individuals of the population, the heuristic used to evaluate each individual, and the artefacts obtained as output.

4.1 BLiMEA

This approach uses a Multi-objective Evolutionary Algorithm (MOEA) with two fitness functions: Information Retrieval (IR), and modification timespan. The consideration of timespans is based on the Defect Localization Principle (DLP). This principle is based on the observation that the most recent modifications to a project are most likely the cause of future bugs [25, 55].

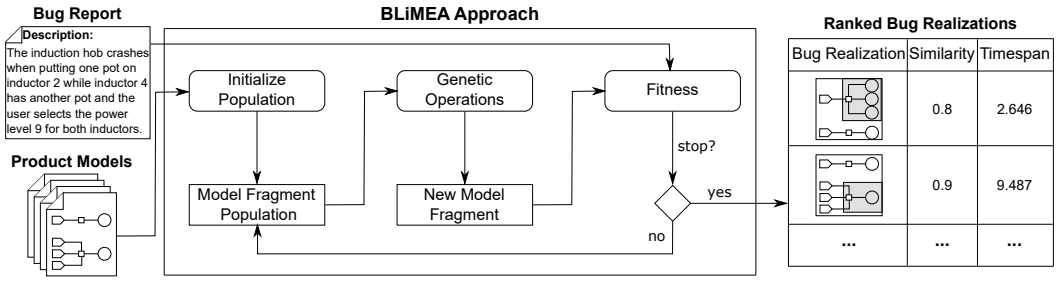

Fig. 4. BLiMEA: Bug Localization in Models with an Evolutionary Algorithm

Fig. 4 presents the BLiMEA approach. The left part shows the inputs for the approach: a bug description and a set of product models. The center shows a simplified representation of the main steps. The 'Initialize Population' step calculates an initial population of model fragments from the input set of product models. The 'Genetic Operations' step produces the new generation of model fragments. Finally, the 'Fitness' step assigns values that assess how good each model fragment is based on: bug description and modification timespan. As output, the approach provides a list of model fragments that might contain the bug.

4.1.1 Input. The approach receives as input a bug description and a set of product models. The bug description usually comes from the textual documentation of a bug report. Therefore, this description will include some domain-specific terms that are similar to those used when specifying the product models. In addition, the software engineers must select a set of product models from the entire family of products and use their knowledge to select those models that contain the bug to be located.

4.1.2 Population. The initialize population step of the approach (see Fig. 4) 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.

In evolutionary algorithms, each individual of the population needs to be encoded so that the genetic operations can be applied to them. This approach uses a binary encoding to represent the individuals (our model fragments). To encode a model fragment, each position of the binary string represents one model element of the parent model. In other words, each individual will either have that position at 0 to indicate that the element is not part of the model fragment or at 1 to indicate that it is part of the model fragment. Thus, we can indicate the subset of elements from the parent model that are part of the model fragment.

4.1.3 Search strategy. The generation of subsequent populations is performed by applying genetic operators. First, a selection operation selects the model fragments that will be used as parents of the new model fragments. Second, other operations are applied to manipulate the model fragments. In other words, new model fragment sets are generated from existing ones through the use of genetic operators: selection, crossover, and mutation.

Selection: This operation selects two individuals that will act as parents for the next population. This approach uses the wheel selection mechanism where fitter individuals are selected more frequently. This mechanism also mitigates premature convergence [1]. In other words, each model fragment from the population has a probability of being selected that is proportional to its fitness score. Therefore, candidates with high fitness values will have higher probabilities of being chosen as parents for the next generation.

Crossover: This crossover operator will take the model fragment from the first parent and the product model from the second parent. Hence, a new model fragment that contains elements from both parents is generated. The operation is performed as follows. First, the model fragment from the first parent is selected. Second, the product model from the second parent is selected. Then, the operation compares the two elements trying to find the model fragment from the first parent in 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 but referencing the product model from the second parent. If the comparison does not find the model fragment from the first parent in the second parent, the operation will return the first parent unchanged.

Mutation: This operation mutates each model fragment in the context of its referenced 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. The type of mutation (addition or removal of elements) is decided randomly. The resulting model fragment will be part of the referenced product model.

The result of the application of these operations is a new model fragment. This new model fragment represents another possible solution that could contain the bug for the specific bug being located.

4.1.4 Assessment. 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 bug description, and the timespan to the most recent model-fragment modifications.

Model-fragment similarity to the bug description: We use an Information Retrieval (IR) technique called Latent Semantic Indexing (LSI) [18] to analyze the relationship between the description of the bug description and the model fragment. LSI works with documents, each of which contains the text that appears in each model fragment. In addition, another document contains the query, which is the relevant text that appears in the bug description. With these documents, our approach constructs a term-by-document co-occurrence matrix. Each of the columns corresponds to each of the documents. Each of the rows corresponds to each of the words extracted from the documents. These words correspond to the names and values of

the properties and methods of each model fragment from the model and the metamodel. The metamodel provides the names of classes, attributes, and methods, while the model provides the values of these attributes and methods. Each cell has the number of occurrences of each of the terms in the model fragments and the query. Once the matrix is built, we normalize and decompose it into a set of vectors using a matrix factorization technique called Singular Value Decomposition (SVD) [32]. 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. Let MF_1 be a model fragment in the population; let V_{MF} be the vector representing the latent semantic value of MF_1, let V_Q be the vector representing the latent semantic value of the query; the angle between V_{MF} and V_Q is θ. The following expression defines the model fragment similarity to the bug description fitness function:

$$similarityFitness(MF_1) = cos(\theta) = \frac{V_{MF} \cdot V_Q}{||V_{MF}|| \cdot ||V_Q||} \quad (1)$$

The most recent model-fragment modification: To apply the Defect Localization Principle, we measure the timespan to the last modification of the model and the metamodel. The timespan is calculated based on the difference between the last modification of a model element (in the model or the metamodel) and the day on which the bug was discovered. Since the modifications affect the model elements, each model element has its modification timespan. Thus, each model element has a constant modification timespan value during the execution of the algorithm, but different model elements have different modification timespan values. A recently modified model element has a lower timespan value than another model element that was modified farther in the past. Since a model fragment is composed of a set of model elements, the timespan weighting of the model fragment depends on the timespan weightings of the model and metamodel elements that compose it. Therefore, we define the timespan value of the model fragment as the value of the most recently modified model or metamodel element. The timespan is based on the number of days and can therefore be very large when the model fragment was modified a long time ago. We used square roots to normalize the timespan because it has achieved good results in other works that use time differences [55]. Let $T = \{t_1, t_2, t_3, ..., t_n\}$ be the timespan set of the model fragment, let $T_{MF} = min(T) = \{t_i \epsilon T | t_i \leq t_j \quad \forall i, j = 1, ..., n\}$ be the minimum value of the timespan set.

$$timespanFitness(MF_1) = \sqrt{T_{MF}} \quad (2)$$

4.1.5 Output. The output of BLiMEA (see Fig. 4) is an ordered set of model fragments that we predict can contain the target bug. The software engineers obtain this set of model fragments, which is intended to identify the parts of the model that are relevant to the bug. To do this, the software engineers can order the ranking following different criteria: the similarity to the bug description, or the most recent model fragment modifications.

4.2 EBRo

This approach uses an evolutionary algorithm guided by a fitness function that considers the similarity to the description of the bug. To measure the textual similarity, we start from an initial model at runtime to which we apply a sequence of reconfigurations, obtaining another model in which we evaluate whether the modified elements are similar to the description of the bug.

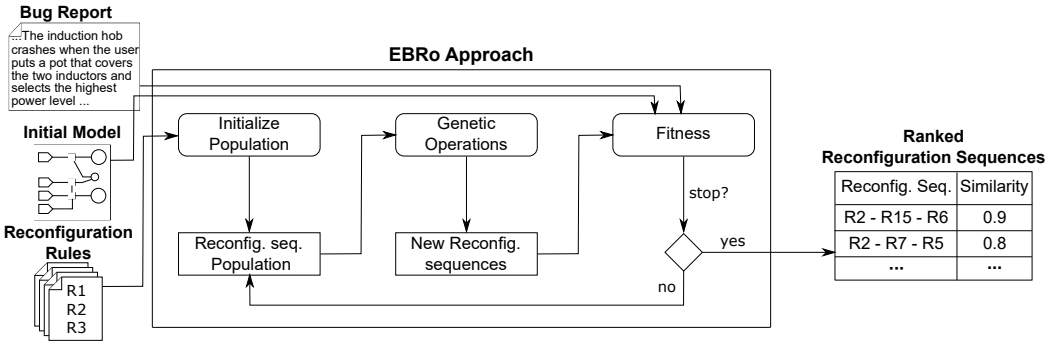

Fig. 5. EBRo: Evolutionary algorithm for Bug Localization in the Reconfigurations of models at runtime

Fig. 5 presents the EBRo approach. The left part shows the inputs for the approach: a bug description, an initial model, and a set of reconfiguration rules. The center shows a simplified representation of the main steps. The 'Initialize Population' step calculates an initial population of reconfiguration sequences from the input set of reconfiguration rules. The 'Genetic Operations' step produces the new generation of reconfiguration sequences. Finally, the 'Fitness' step assigns values that assess how good each reconfiguration sequence is based on the similarity to the bug description. As output, the approach provides a list of reconfiguration sequences that might trigger the bug.

4.2.1 Input. The approach receives as input a set of reconfiguration rules, an initial model, and a bug description. The set of reconfiguration rules describes the changes in the model at runtime. The reconfiguration rules are triggered by context changes. The initial model is the model that specifies the initial configuration. Finally, the bug description of the bug that we want to locate is written using natural language.

4.2.2 Population. Each individual of the population is a reconfiguration sequence. To represent each individual, we use a vector representation. Each vector's dimension represents a reconfiguration rule. Thus, a solution is defined as a sequence of reconfigurations applied to a model. The size of the solution represents the number of reconfigurations (dimensions) in the vector. When created, the order of the reconfigurations corresponds to their positions in the vector.

4.2.3 Search strategy. As in the previous section, to allow the generation of new populations, we have to define a selection of the individuals that will work as parents of the new populations. In addition, we have to define the operators that allow the creation of the new individuals of the population: crossover and mutation.

Selection: To select individuals, we use stochastic universal sampling (SUS) [7]. This technique of selection of an individual is directly proportional to its relative fitness in the population. SUS is a random selection algorithm that gives a higher probability of selection to the fittest solutions while still giving a chance to every solution. In each iteration of the algorithm, SUS is used to select individuals from the population (P_n) for the next generation of the population (P_{n+1}). The selected individuals will be the ones that generate the next individuals using genetic operations.

Crossover: We use a single, random, cut-point crossover. It starts by selecting and splitting two parent solutions at random. When two parent individuals are selected, a random cut

point is determined to split them into two sub-vectors. Then, the crossover creates two child solutions: for the first child, putting the first part of the first parent with the second part of the second parent; and, for the second child, putting the first part of the second parent with the second part of the first parent. Each solution has a length limit in terms of the number of reconfigurations. When applying the crossover operator, the new solution may have the minimum length between the two parents. Thus, the crossover operator must enforce the length limit constraint by eliminating some reconfiguration rules.

Mutation: This operator consists of randomly changing one or more reconfigurations in the vector of reconfigurations. Given an individual, the mutation operator first randomly selects some positions in the vector representation of the individual. Then, the selected dimensions are replaced by other reconfiguration rules.

As a result, new reconfiguration sequences are created. In other words, the new reconfiguration sequences represent other possible solutions that can trigger the specific bug being located.

4.2.4 Assessment. To assess each of the proposed reconfiguration sequences, we use LSI, which is the same information retrieval technique presented in the BLiMEA approach. The input to perform the assessment is a set of reconfiguration sequences, and the output is the set of reconfiguration sequences, where each reconfiguration sequence has been assigned with a fitness value regarding its similarity to the bug description.

Our EBRo approach applies the following steps to extract the texts used by LSI. First, we apply the reconfiguration sequence to the initial model configuration. After applying it, we obtain a new model from which we extract the model elements that have been modified by the reconfigurations. The texts for the LSI documents are the names and values of the properties and methods of each model element. In this approach, the LSI documents are model elements. In other words, a document of text is generated from the text of the model elements that have been modified by the reconfiguration. The query is constructed from the text that appears in the bug description.

The union of all the keywords extracted from the documents (model elements) from the query (bug description) are the terms (rows) used by our LSI fitness. Each column is one of the model elements that have been modified by the reconfiguration. The last column is the query obtained from the bug description of the user. Each row is one of the terms extracted from the text composed by all of the model elements and the query itself. Each cell has the number of occurrences of each of the terms in the model elements. Once the matrix is built, we normalize and decompose it into a set of vectors using the same matrix factorization technique as in the BLiMEA approach (see Section 4.1.4).

4.2.5 Output. The output of EBRo (see Fig. 5) is an ordered set of reconfiguration sequences that we predict can trigger the target bug. The ranking is ordered following the similarity to the bug description.

A detailed explanation of both approaches and the algorithms can be found in [3, 4].

4.3 CoEB: Coevolutionary Algorithm based on BLiMEA and EBRo

In Coevolutionary algorithms, instead of evolving a population of individuals that represents the solution, it is more appropriate to co-evolve subpopulations of individuals representing specific parts of the solution [42]. There are two types of coevolution: cooperation and competition. Taking into account the problem we want to solve; we adapt a competitive solution based on BLiMEA and

EBRo where the fitness value of an individual depends on the fitness value of other individuals from the other subpopulation.

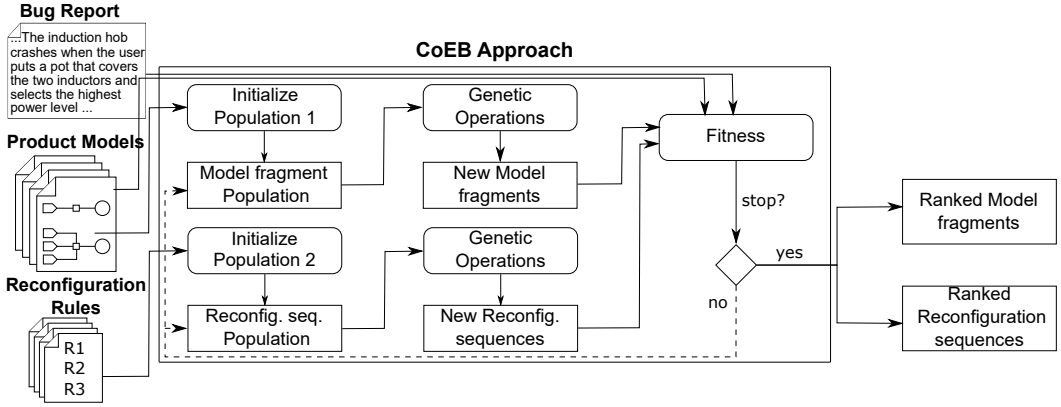

Fig. 6. CoEB: Coevolutionary Algorithm based on BLiMEA and EBRo

Fig. 6 presents the CoEB approach. The left part shows the inputs for the approach: a bug description, a set of product models, and a set of reconfiguration rules. The center shows a simplified representation of the main steps. The 'Initialize Population' step calculates the two initial populations. The 'Genetic Operations' step produces the new generations of both populations. Finally, the 'Fitness' step assigns values that assess how good each individual of each population is. As output, the approach provides a list of model fragments that might contain the bug and a list of reconfiguration sequences that might trigger the bug.

4.3.1 Input. The approach receives the same inputs as the previous approaches: a bug description, a set of product models, and a set of reconfiguration rules. One of the product models will work as the initial model in the same way as in EBRo (see Section 4.2.1).

4.3.2 Population. The first population is composed of model fragments. The initial set of model fragments is randomly extracted from the product models. The approach uses the same encoding as BLiMEA (see Section 4.1.2). The second population is composed of reconfiguration rules. The approach uses the same encoding as EBRo (see Section 4.2.2).

4.3.3 Search strategy. To allow the generation of new populations, we use the techniques described in the previous approaches. For the model fragment population, we apply the techniques presented in BLiMEA. To select individuals, we use the wheel selection mechanism. To perform the crossover, we take a model fragment from one parent and the product model from the second parent. To perform the mutation, we mutate each model fragment in the context of its referencing product model by adding or removing elements (see Section 4.1.3).

For the reconfiguration rules population, we apply the techniques used in EBRo. To select individuals, we use stochastic universal sampling [7]. To perform the crossover, we use a single, random, cut-point crossover. To perform the mutation, we randomly change one or more reconfigurations in the vector of reconfigurations (see Section 4.2.3).

4.3.4 Assessment. To assess each of the individuals, we use different fitness functions for each subpopulation.

Fitness functions of the first population. To evaluate the model fragments for the first population, the fitness function is based on: 1) model-fragment similarity to the bug description, 2) the most recent model-fragment modification, and 3) maximizing the coverage of the fittest reconfiguration sequences in the generation.
The first two fitness functions of the first population, which correspond to model-fragment similarity to the bug description and the most recent model-fragment modification, are the same as the ones described in BLiMEA (see Section 4.1.4). For the third fitness function, we take the first three reconfiguration sequences that have the best similarity values. Then, we try to apply the sequences to each model fragment. The more reconfiguration rules that can be applied, the higher the fitness value. In order to normalize the numeric value, we define this value as the percentage of reconfiguration rules applied of the total number of the reconfiguration rules of the three reconfiguration sequences. Thus, we can get values in the range of $[0, 1]$.

Fitness functions of the second population. To evaluate the reconfiguration sequences for the second population, which is executed in parallel, the fitness function is based on: 1) similarity to the bug description, and 2) maximizing the number of model elements of the fittest models of the generation in which we can apply the reconfiguration sequence.
The fitness function of the second population, which corresponds to similarity to the bug description, is the same as the one presented in EBRo (see Section 4.2.4). For the second fitness function, we take the first three model fragments that have the best similarity values and the most recent timespan modification. Once we have the model fragments, we take the product models that correspond to these model fragments. Then, we try to apply the sequences to each product model. The more model elements modified by the reconfiguration rules that appear in the three model fragments, the higher the fitness value. Similar to the third fitness function of the first population, to normalize the numeric value, we define this value as the percentage of model elements modified of the total number of model elements of the model fragments. Thus, we can get values in the range of $[0, 1]$.

4.3.5 Output. The output of CoEB is an ordered set of model fragments and an ordered set of reconfiguration rules that we predict can lead to the target bug. The software engineers can order the rankings based on different fitness functions.

4.4 Hill Climbing

The fourth algorithm that we use is the steepest ascent hill climbing with replacement [34]. We adapt the hill-climbing algorithm presented by Font et al. [20] for feature location in models.

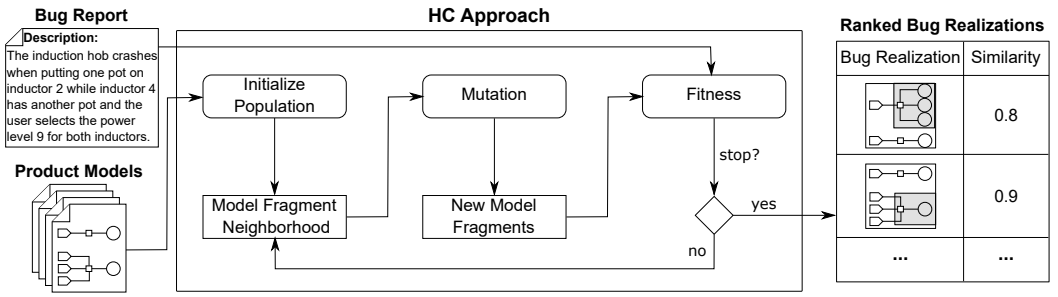

Fig. 7. HC: Hill Climbing for Bug Localization in Models

Fig. 7 presents the HC approach. The left part shows the inputs for the approach: a bug description and a set of product models. The center shows a simplified representation of the main steps. The 'Initialize Population' step calculates an initial model fragment from the input set of product models. The 'Mutation' step produces a new generation of model fragments. Finally, the 'Fitness' step assigns values that assess how good each model fragment is based on the similarity to the bug description. As output, the approach provides a list of model fragments that might contain the bug.

4.4.1 Input. The approach receives the same input as the previous approach, BLiMEA: a bug description and a set of product models (see Section 4.1.1)

4.4.2 Population. The algorithm starts with a random individual, i.e., a random model fragment that is extracted from the product models. Similar to the BLiMEA approach, this approach uses a binary encoding to represent the individual (see Section 4.1.2)

4.4.3 Search strategy. This approach requires the generation of a neighborhood. The neighborhood is created based on the initial model fragment. The operation that is used to create the neighbors is the mutation operation described in BLiMEA (see Section 4.1.3). After exploring the neighborhood, if a neighbor has a better fitness value than the current model fragment, the search moves to that model fragment. The selected model fragment will be the one used to generate the new neighborhood.

4.4.4 Assessment. To assess each individual, we use the model-fragment similarity to the bug description described in BLiMEA (see Section 4.1.4). The quality of the solution found by this algorithm greatly depends on the first individual created because the HC algorithm does not have random restarts.

4.4.5 Output. The output of HC is an ordered set of model fragments that can contain the bug that we want to locate. The set is ordered based on the similarity to the bug description.

4.5 Random Search (RS)

We use a random search algorithm as a sanity check. If RS outperforms an intelligent search method, we can conclude that there is no need to use a metaheuristic search.

The algorithm starts with a random initial model and a random reconfiguration sequence. Then, a new random reconfiguration sequence is generated. The search moves to a new reconfiguration sequence if the fitness value is better than the current best reconfiguration sequence. The loop is repeated until the stop condition is met.

4.6 Manual refinement of the output

The outputs of the approaches are ranked lists. In the case of BLiMEA, the list can be ordered based on the similarity to the bug report or the most recent model modifications. In the case of EBRo, the ranking is ordered based on the similarity to the bug report. For CoEB, the lists can be ordered based on the different fitness functions used. For HC, the ranking is ordered based on the similarity to the bug report. These rankings are shown to the software engineers, and they can then choose the one that they believe is responsible for the bug.

In this work, we allow the software engineers the ability to modify the solution obtained. In other words, once the software engineers get the solution, they can manually refine the model fragment or the reconfiguration sequence.

In the case of model fragments, they can add model elements to the model fragment or remove model elements from the model fragment. As an example, we want to locate the model fragment that corresponds to the bug description entitled *"The secondary inductor in double hotplate crashes"*.

One of the model fragments proposed by BLiMEA contains the primary and the secondary inductors and the power manager with the channels that connect them. When the software engineers see this fragment, they can decide whether or not the channel that connects the secondary inductor to another power group is relevant. If it is relevant, then they can add it to the model fragment; if the power manager is not relevant for this bug, they can remove it from the model fragment.

In the case of reconfiguration sequences, they can add a reconfiguration to the sequence, remove a reconfiguration from the sequence, or change one reconfiguration for another. As an example, we want to locate the reconfiguration sequence that triggers the bug mentioned above. One of the reconfiguration sequences contains three reconfiguration rules (*R1: newChannelForDoubleInductor*; *R2: redirectPowerManagerFromInverter*; and *R3: disconnectFromPowerManager*). When the software engineers see this sequence, they can decide if the rule *R4: activatePowerFromInverter* is relevant for this bug. If it is relevant, they can add it to the reconfiguration sequence in any position. Other options are, for example, if the rule *R2* is not relevant for this bug, then they can remove it from the reconfiguration sequence, or if the rule *R3* should occur before the rule *R2*, they can modify the order of the reconfiguration sequence.

4.7 Implementation details

For comparison, we implement our single-objective evolutionary algorithm and our multi-objective evolutionary algorithm with the same architecture, as performed in other research works [28]. Our algorithms are based on NSGA-II [17], which is one of the most frequently used multi-objective evolutionary algorithms. In BLiMEA, given a set of model fragments where each model fragment has a fitness value for its bug similarity and its recent time modifications (see Section 4.1.4), NSGA-II orders these model fragments by means of non-dominated sorting. A model fragment is non-dominated if the following hold: 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. Similarly, in EBRo, given a set of reconfiguration sequences where each reconfiguration sequence has a fitness value for its bug similarity (see Section 4.2.4), NSGA-II orders these reconfiguration sequences by means of non-dominated sorting. A reconfiguration sequence is non-dominated if the following hold: 1) there is no other reconfiguration sequence that is better than the current one for some fitness value; and 2) the current reconfiguration sequence does not worsen other fitness values. As a result, NSGA-II finds Pareto-optimal sequences of reconfiguration rules. In the same way, NSGA-II orders the model fragments and the reconfiguration rules in the CoEB approach (see Section 4.3).

We performed some parameter tuning to find the best values for the parameters of our algorithm. However, 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. Therefore, we have principally chosen values for those settings that are commonly used in the literature [39]. Default values are good enough to measure the performance of search-based techniques in the context of testing [6]. Hence, the crossover operation is applied with a crossover probability (pc) of 0.9, and the mutation operation is applied with a probability (pm) of 0.1.

The number of generations (repetitions of the genetic operations and fitness loop) that is 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). 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 s.

5 HOW WE COMBINE THE BLIMEA AND EBRO APPROACHES

The BLiMEA approach covers bug localization in Model-based systems, while the EBRo approach covers only the Reconfigurable models at run-time systems (see Fig. 3). In addition, the artefacts used by each approach are different. BLiMEA uses bug reports, the metamodel, and design time models, and EBRo uses bug reports, design-time models, run-time models, and reconfiguration rules (see Table 1). The combination of EBRo and BLiMEA covers the bug localization in Reconfigurable models at run-time systems and takes into account the artefacts that can be present in these systems: bug reports, the metamodel, design-time models, run-time models, and reconfiguration rules.

Fig. 8 depicts how we can combine our EBRo and BLiMEA approaches. After executing the EBRo approach, we obtain a list of reconfiguration sequences. These reconfiguration sequences can be applied to the initial model (used as input by EBRo). When we apply the reconfiguration sequences to the initial model, we obtain one new model for each reconfiguration sequence. In other words, we obtain new models modified by these reconfigurations. These new models can be used as input by the BLiMEA approach. Finally, the output of the combination of EBRo and BLiMEA is a set of model fragments that is derived from the models obtained with the reconfiguration sequences. This set of model fragments can be ordered based on the similarity to the bug description or based on the most recent model fragment modifications.

On the other hand, both the reconfiguration sequences and the model fragments can be manually refined (see 4.6). After the execution of each approach, humans can modify them with the motivation of correctly identifying elements that are relevant to the bug. In the example of Fig. 8, the EBRo approach is executed before the BLiMEA approach. When EBRo finishes the execution, the output is a set of reconfiguration rules. The software engineers (humans) can inspect the output. If they recognize something missing or something that they know is wrong, they can modify each reconfiguration rule from the set outputted. Then, BLiMEA uses the set of model fragments that is derived from the models obtained with the reconfiguration sequences. The more precise the set of input models, the more precise the solution obtained, therefore, the refinement of the output, in this case, the reconfigurations, by humans can be very relevant. When BLiMEA finishes the execution, the output is a set of model fragments. Again, the software engineers can inspect the output and modify each model fragment from the set outputted. This is important because before fixing a bug, it must be identified. Therefore, the more precise the result obtained, the easier it is to find the relevant artefacts for the bug.

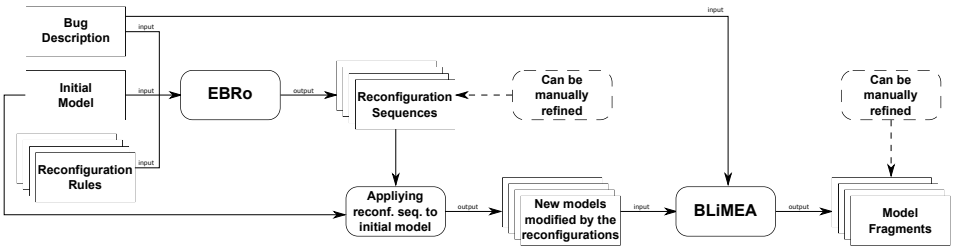

Fig. 8. The combination of EBRo and BLiMEA

5.1 Different combinations of the approaches

When considering the possible combinations of our approaches, we have taken into account the following: 1) we have four different approaches for bug localization; 2) the EBRo approach is the more specific for locating bugs in systems with reconfigurable models at runtime; and 3) the software engineers (humans) can refine the results obtained in each of the approaches.

Taking into account these points, we are going to combine the approaches, obtaining a total of ten variants.

(1) EBRo.
(2) BLiMEA.
(3) EBRo and then BLiMEA: we apply BLiMEA on the solution obtained in EBRo (EB).
(4) EBRo, manual refinement, and BLiMEA: we apply BLiMEA on the manually refined solution obtained in EBRo (EmB).
(5) EBRo, BLiMEA, and manual refinement: we apply BLiMEA on the solution obtained in EBRo, and then the solution is manually refined (EBm).
(6) EBRo, manual refinement, BLiMEA, and manual refinement: we apply BLiMEA on the manually refined solution obtained in EBRo, and then the solution is manually refined (EmBm).
(7) CoEB.
(8) CoEB and manual refinement: we apply CoEB, and the solution is manually refined (CoEBm).
(9) HC.
(10) HC and manual refinement: we apply HC, and the solution is manually refined (HCm).

6 EVALUATION

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

To evaluate the approaches, we applied them to two industrial case studies from our industrial partners, BSH, a leading manufacturer of home appliances in Europe, and CAF, an international provider of railway solutions all over the world. There are two aspects that we want to evaluate regarding the use of our approaches and the way of involving humans in the process. In order to address the evaluation of these aspects, we formulated the following two research questions:

- RQ_1**:** What is the impact of using the two artefacts simultaneously (design models and reconfigurations) on the quality of the results?
- RQ_2**:** What is the impact of manual refinement on the quality of the results?

Answering RQ_1 allows us to compare the performance results (in terms of recall, precision, the F-measure, and Matthews Correlation Coefficient, MCC) of each of our approaches separately and combined. In addition, we compare the approaches 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. Answering RQ_2 with the same metrics allows us to know if involving humans in the process of the approaches improves the results even more.

6.1 Case Study and Oracle preparation

To evaluate the approach, we applied it to two industrial case studies from our industrial partners: BSH and CAF. For each of the case studies, we prepared the oracles. The oracle is the ground truth and is used to compare the results provided by the approaches and the random search (RS). 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 and reconfiguration rules that contain or trigger the target bugs.

6.1.1 BSH. The first case study where we applied our evaluation process is the Induction Hob Product Family of our industrial partner BSH (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 representative of the bugs that occur in BSH.

For each of the 37 bugs, we created the test cases needed as input by the different approaches. One of them includes the set of product models where that bug was manifested and a bug description for BLiMEA and HC. The second one includes the initial model, the set of reconfiguration rules, and a bug description for EBRo. The third one includes the set of product models, the set of reconfiguration rules, and a bug description. All of them were obtained from the documentation. An example of the models and model fragments of BSH can be seen in Fig. 1 in Section 2. More details about the models and model fragments of BSH are available at: https://www.youtube.com/watch?v=nS2sybEv6j0.

6.1.2 CAF. The second case study where we applied our evaluation process is a family of Programmable Logic Controller (PLC) software that is used to manage the trains they manufacture, which has been under development for more than 25 years. Their trains are found all over the world in different forms: regular trains, subway, light rail, monorail, etc. A train is composed of several pieces of equipment throughout its vehicles and cabins which are designed and manufactured by different suppliers. The equipment performs specific tasks for the train, e.g., the traction equipment, the compressors that feed the brakes, the pantograph that collects energy from the overhead wires, or the circuit breaker that isolates or connects the electrical circuits of the train. The software of the train is in charge of making all of the equipment cooperate to achieve the train's functionality, ensuring compliance with the specific regulations of each country.

The DSL of our industrial partner has the expressiveness necessary to describe the interaction between the main equipment installed in a train unit and the reconfigurations at runtime. Examples of run-time reconfigurations can be coupling (when two trains are coupled to increase transport capacity or to rescue a train that is damaged) or converter assistance (allowing current to pass from a converter to the equipment assigned to peers in case of overload or system failure).

The oracle extracted from CAF is composed of 23 trains, which, on average, are composed of more than 1,200 elements. Our industrial partner provided us with documentation of 56 bug reports and the approved model fragments that contain the bugs or the approved reconfiguration sequences that trigger the bugs. The approved model fragments have an average of 19 model elements. As in the BSH case study, each model element has properties (that include terms) and a modification timespan, which are used to differentiate among model elements. Again, six domain engineers from our industrial partner were involved in providing the set of 56 bugs. The set of bugs provided is the most representative of the bugs that occur in CAF combining importance and frequency. In the same way as in the previous case study, for each of the 56 bugs, we created the test cases needed as input by the different approaches, all of which were obtained from the CAF documentation.

Since the software models of the case studies are currently operating or will be released in the near future, this information is limited by confidentiality agreements with our industrial partners. Nevertheless, for purposes of replicability, the CSV files used as input in the statistical analysis have been attached to the submission system. We have also attached a document with the R-scripts that were implemented to analyze the results. The files are published online at https://svit.usj.es/TOSEM_Arcega_evaluation/.

6.2 Experimental setup

This experiment tries to answer the two research questions presented. Fig. 9 shows an overview of the process that was followed to evaluate our approaches. 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 the 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 provided by our industrial partner are the models of the induction hobs, the entire set of reconfiguration rules, and the bug reports.

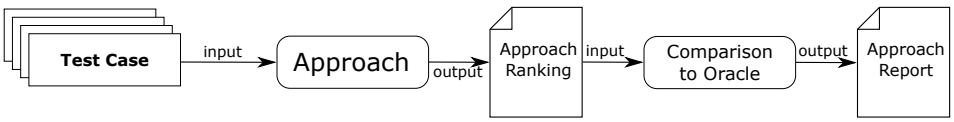

Fig. 9. Evaluation process

Each time that we run an approach, we obtain a set of results for a bug. As the approaches perform 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 [5]. Therefore, for the BSH case study, we executed 30 independent runs for each of the 37 test cases for each approach, i.e., 37 (bugs) x 10 (approaches) x 30 repetitions = 11,100 independent runs. For the CAF case study, we executed 30 independent runs for each of the 56 test cases for each approach, i.e., 56 (bugs) x 10 (approaches) x 30 repetitions = 16,800 independent runs.

To establish the stop condition, we perform a preliminary study to determine the time needed for the evolutionary algorithm to converge (the point where there are no changes in further generations). For the localization performed in these domains, the time needed was below 60 seconds for all of the bugs; therefore, we established the stop condition at 80 seconds (adding a margin to ensure convergence). When we combine two approaches, we established the stop condition at 160 seconds (80 seconds for each approach).

To perform manual refinement, two software engineers from each industrial partner were involved. The engineers are domain experts of the BSH induction hobs division and the CAF train controller software division. These engineers were chosen randomly from all of those who make up the teams.

In the evaluation, one of the engineers performs the manual refinement and the other engineer confirms the modifications. In other words, the engineers act as one step more of the approach under evaluation. They have the opportunity to fix the parts that they believe are wrong. The first engineer can modify the solutions obtained as explained in Section 4.6. The second engineer accepts or rejects the modifications. If the second engineer rejects the modifications, the first engineer can modify the solution again. The engineers perform between two and seven modifications to the solutions obtained (on average, four modifications in each solution).

After running the approaches, in order to compare them with each other and the RS approach, we take the best solutions from each of the approaches for each of the bugs (the first solution in the ranking) as suggested in [28]. Then, we compare them to the actual solution (from the oracle) that contains the model fragment or the trigger of the target bug in order to get a confusion matrix.

A confusion matrix is a table that allows the visualization of the performance of a classification algorithm. In our case, each solution that is output by the EBRo and the RS approaches is a sequence of reconfigurations composed of a subset of reconfigurations that are present in the sequence that

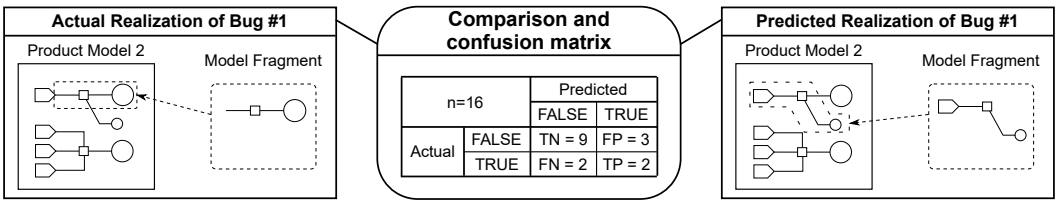

Fig. 10. Example of the comparison process and the confusion matrix

triggers the bug, while each solution that is output by the BLiMEA, CoEB, and HC 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 or reconfigurations, the presence or absence of each model element or reconfiguration will be considered as a classification. Therefore, our confusion matrices will distinguish between two values (TRUE or presence and FALSE or absence).

Fig. 10 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 best solutions predicted by each of the approaches for each of the bugs. The left part of Fig. 10 shows the actual model fragment that contains the bug (obtained from the oracle and considered the ground truth), while the right part of Fig. 10 shows the predicted model fragment output by the approach. The confusion matrix arranges the results of the comparison into four categories:

- True positive (TP): an element present in the predicted solutions that is also present in the actual solution,
- True Negative (TN): an element not present in the predicted solution that is not present in the actual solution,
- False Positive (FP): an element present in the predicted solution that is not present in the actual solution, and
- False Negative (FN): an element not present in the predicted solution that is present in the actual solution.

The confusion matrix holds the results of the comparison between the predicted results and the actual results. The result of the sum of all of 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 the approaches.

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

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

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

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

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

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5}$$

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 1 (i.e., all of 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 1 (i.e., all of the model fragments from the solution are present in the model fragment that contains the bug from the oracle). A value of 1 in precision and 1 in 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 that 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 [41]. It is defined as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}} \tag{6}$$

6.3 Results

In this section, we present the results obtained for each case study in EBRo, BLiMEA, CoEB, HC, the combinations, and the RS approaches in BSH and CAF. Table 2 shows the mean values of recall, precision, the F-measure, and MCC for the two research questions. All of the algorithms obtained better results than the random search. For the first research question (the top part of the tables), the EBRo+BLiMEA (EB) approach obtained the best results in recall, precision, and MCC, providing an average value of 0.83 in recall, 0.73 in precision, and 0.75 in MCC in BSH, and 0.78 in recall, 0.70 in precision, and 0.70 in MCC in CAF. For the second research question (the bottom part of the tables), the EBRo+BLiMEA+manual refinement (EBm) approach obtained the best results in recall, precision, and MCC, providing an average value of 0.90 in recall, 0,80 in precision, and 0.82 in MCC in BSH, and 0.87 in recall, 0.78 in precision, and 0.80 in MCC in CAF. Overall, in terms of recall, precision, and MCC, EBm outperformed the rest of the approaches.

6.4 Statistical Analysis

To properly compare our approaches, all of the data resulting from the empirical analysis was analyzed using statistical methods following the guidelines in [5]. The goals of our statistical analysis are 1) to provide formal and quantitative evidence (statistical significance) that the approaches do 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).

6.4.1 Statistical Significance. To enable statistical analysis, all of the algorithms should be run a large enough number of times (separately) 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

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

		BSH			
		Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)	MCC ± (σ)
RQ1	EBRo	0.81 ± 0.08	0.69 ± 0.08	0.74 ± 0.06	0.71 ± 0.06
	BLiMEA	0.77 ± 0.11	0.65 ± 0.07	0.70 ± 0.06	0.67 ± 0.07
	EB	0.83 ± 0.13	0.73 ± 0.11	0.77 ± 0.08	0.75 ± 0.09
	CoEB	0.69 ± 0.12	0.60 ± 0.11	0.63 ± 0.07	0.60 ± 0.08
	HC	0.41 ± 0.06	0.47 ± 0.09	0.43 ± 0.05	0.37 ± 0.06
	Random Search	0.35 ± 0.04	0.41 ± 0.07	0.37 ± 0.04	0.31 ± 0.04
RQ2	EmB	0.71 ± 0.12	0.59 ± 0.10	0.63 ± 0.07	0.59 ± 0.08
	EBm	0.90 ± 0.09	0.80 ± 0.12	0.84 ± 0.08	0.82 ± 0.09
	EmBm	0.73 ± 0.09	0.64 ± 0.12	0.67 ± 0.08	0.64 ± 0.09
	CoEBm	0.69 ± 0.09	0.71 ± 0.11	0.69 ± 0.07	0.66 ± 0.08
	HCm	0.43 ± 0.05	0.52 ± 0.09	0.46 ± 0.05	0.41 ± 0.07
		CAF			
		Recall ± (σ)	Precision ± (σ)	F-measure ± (σ)	MCC ± (σ)
RQ1	EBRo	0.76 ± 0.08	0.65 ± 0.06	0.70 ± 0.05	0.66 ± 0.06
	BLiMEA	0.77 ± 0.10	0.63 ± 0.07	0.69 ± 0.07	0.65 ± 0.08
	EB	0.78 ± 0.05	0.70 ± 0.06	0.74 ± 0.05	0.70 ± 0.06
	CoEB	0.68 ± 0.12	0.60 ± 0.10	0.62 ± 0.07	0.59 ± 0.08
	HC	0.36 ± 0.05	0.46 ± 0.08	0.40 ± 0.04	0.34 ± 0.05
	Random Search	0.33 ± 0.05	0.39 ± 0.07	0.35 ± 0.04	0.28 ± 0.06
RQ2	EmB	0.68 ± 0.11	0.58 ± 0.11	0.62 ± 0.09	0.58 ± 0.11
	EBm	0.87 ± 0.10	0.78 ± 0.07	0.82 ± 0.07	0.80 ± 0.08
	EmBm	0.72 ± 0.08	0.61 ± 0.10	0.66 ± 0.07	0.62 ± 0.08
	CoEBm	0.64 ± 0.07	0.65 ± 0.10	0.64 ± 0.06	0.60 ± 0.07
	HCm	0.41 ± 0.05	0.49 ± 0.08	0.44 ± 0.05	0.39 ± 0.05

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. A low $p - Value$ suggests that the sample provides enough evidence that the null hypothesis H_0 can be rejected. It is accepted by the research community that a $p - Value$ under 0.05 is statistically significant [5], 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 [22]. In addition, according to Conover [15], the Quade test has shown better results than the others when the number of algorithms is low (no more than four or five algorithms).

The $p - value$ obtained in the test is $\ll 2.2x10^{-16}$ for recall, precision, and MCC, for the two research questions; the statistics values obtained are 28.782, 28.524, and 42.878 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 that there are differences among the algorithms for the performance indicators of recall, precision, and MCC.

Table 3. Holm's post hoc $p - Values$ for each pair of algorithms in BSH and CAF

			Recall	Precision	MCC
BSH	RQ1	EBRO vs. BLiMEA	0.58898	0.34266	0.20577
		EBRO vs. EB	0.58898	0.38534	0.08363
		EBRO vs. CoEB	**0.03605**	**0.00535**	**0.00019**
		EBRO vs. HC	$\mathbf{2.5x10^{-15}}$	$\mathbf{3.6x10^{-13}}$	$\mathbf{5.1x10^{-16}}$
		EBRO vs. RS	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$
		BLiMEA vs. EB	0.10281	0.06287	**0.00545**
		BLiMEA vs. CoEB	0.37913	0.34266	**0.01234**
		BLiMEA vs. HC	$\mathbf{1.9x10^{-12}}$	$\mathbf{4.1x10^{-09}}$	$\mathbf{1.2x10^{-12}}$
		BLiMEA vs. RS	$\mathbf{7.8x10^{-16}}$	$\mathbf{5.1x10^{-13}}$	$\mathbf{\ll 2.2x10^{-16}}$
		EB vs. CoEB	**0.00061**	**0.00024**	$\mathbf{1.6x10^{-08}}$
		EB vs. HC	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{1.7x10^{-15}}$	$\mathbf{\ll 2.2x10^{-16}}$
		EB vs. RS	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$
		CoEB vs. HC	$\mathbf{2.0x10^{-08}}$	$\mathbf{1.8x10^{-05}}$	$\mathbf{8.1x10^{-06}}$
		CoEB vs. RS	$\mathbf{1.7x10^{-11}}$	$\mathbf{8.3x10^{-09}}$	$\mathbf{1.0x10^{-10}}$
		HC vs. RS	0.58898	0.34266	0.08236
	RQ2	EB vs. EmB	$\mathbf{3.8x10^{-06}}$	$\mathbf{8.8x10^{-07}}$	$\mathbf{3.8x10^{-10}}$
		EB vs. EBm	0.058	**0.0379**	0.062
		EB vs. EmBm	$\mathbf{4.4x10^{-05}}$	**0.0037**	$\mathbf{5.7x10^{-06}}$
		EmB vs. EBm	$\mathbf{1.7x10^{-10}}$	$\mathbf{1.4x10^{-11}}$	$\mathbf{7.4x10^{-15}}$
		EmB vs. EmBm	0.512	**0.0379**	0.062
		EBm vs. EmBm	$\mathbf{3.7x10^{-09}}$	$\mathbf{5.2x10^{-07}}$	$\mathbf{3.8x10^{-10}}$
		CoEB vs. CoEBm	0.98	**0.0013**	**0.0032**
		HC vs. HCm	0.11	0.13	**0.016**
CAF	RQ1	EBRO vs. BLiMEA	0.9557	0.10840	0.4278
		EBRO vs. EB	1	0.15854	0.0837
		EBRO vs. CoEB	0.0713	**0.03849**	**0.0017**
		EBRO vs. HC	$\mathbf{5.1x10^{-13}}$	$\mathbf{1.8x10^{-12}}$	$\mathbf{3.2x10^{-15}}$
		EBRO vs. RS	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$
		BLiMEA vs. EB	1	**0.00089**	**0.0149**
		BLiMEA vs. CoEB	**0.0045**	0.55986	**0.0165**
		BLiMEA vs. HC	$\mathbf{1.2x10^{-15}}$	$\mathbf{1.9x10^{-07}}$	$\mathbf{3.9x10^{-13}}$
		BLiMEA vs. RS	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{7.8x10^{-13}}$	$\mathbf{\ll 2.2x10^{-16}}$
		EB vs. CoEB	**0.0126**	**0.00010**	$\mathbf{1.3x10^{-07}}$
		EB vs. HC	$\mathbf{1.1x10^{-14}}$	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$
		EB vs. RS	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$	$\mathbf{\ll 2.2x10^{-16}}$
		CoEB vs. HC	$\mathbf{3.8x10^{-07}}$	$\mathbf{2.9x10^{-06}}$	$\mathbf{2.2x10^{-06}}$
		CoEB vs. RS	$\mathbf{4.6x10^{-12}}$	$\mathbf{2.1x10^{-11}}$	$\mathbf{6.2x10^{-11}}$
		HC vs. RS	0.1521	0.09763	0.0875
	RQ2	EB vs. EmB	**0.00066**	$\mathbf{7.5x10^{-05}}$	$\mathbf{3.4x10^{-06}}$
		EB vs. EBm	**0.00066**	**0.0002**	$\mathbf{5.7x10^{-06}}$
		EB vs. EmBm	0.10110	**0.0167**	**0.0087**
		EmB vs. EBm	$\mathbf{3.6x10^{-11}}$	$\mathbf{3.6x10^{-13}}$	$\mathbf{\ll 2.2x10^{-16}}$
		EmB vs. EmBm	0.10110	0.0763	**0.0226**
		EBm vs. EmBm	$\mathbf{2.3x10^{-07}}$	$\mathbf{2.5x10^{-09}}$	$\mathbf{9.6x10^{-12}}$
		CoEB vs. CoEBm	0.15	**0.017**	0.28
		HC vs. HCm	$\mathbf{7.7x10^{-05}}$	0.16	**0.0012**

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 of the 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 algorithms. 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, in the BSH case study, when we compare EBRo with BLiMEA, EB, and CoEB (the first, second, and third rows), BLiMEA with EB and CoEB (the sixth and seventh rows), HC with RS (the fifteenth row), EB with EBm (the seventeenth row), EmB with EmBm (the nineteenth row), CoEB with CoEBm (the twenty-first row), or HC with HCm (the twenty-first row), the values obtained for all or some of the parameters are greater than the threshold. The results are similar in the CAF case study; all of these comparisons obtain values that are greater than the threshold in all or some of the parameters. This indicates that the differences between those algorithms could be due to the stochastic nature of the algorithms and are not significant.

6.4.2 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 [5]. 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 $\hat{A}_{12}$ [24, 49]. $\hat{A}_{12}$ 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 algorithm of the two algorithms compared, and $\hat{A}_{12} = 0.3$ means that we would obtain better results in 70% of the runs with the second algorithm of the two algorithms compared. Thus, we have an $\hat{A}_{12}$ value for every pair of algorithms.

Table 4 shows the values of the effect size statistics. The left part shows the $\hat{A}_{12}$ value for each pair of algorithms. In general, the largest differences were obtained between EBm and EmB, and between EBm and EmBm (where EBm achieves better results almost every time). When comparing EBm and EB, the differences are not so large; EBm achieved better results around 70% of the times. When comparing CoEB and CoEBm, the approach with manual refinement obtains better results around 70% of the times for precision and MCC. However, this difference is lower for recall, where CoEB and CoEBm are equivalent. When comparing HC and HCm, the approach with manual refinement obtains better results around 65% of the times. The right part shows the $\hat{A}_{12}$ value for each bug, where N indicates the number of bugs for which the difference is significant and Mean± (σ) indicates the $\hat{A}_{12}$ average value and the standard deviation. In general, almost all bugs show significant differences. However, when we compare EB and EBm in the CAF case study, few bugs present significant differences: two in recall, five in precision, and three in MCC.

The combination of the EBRo approach and the BLiMEA approach plus manual refinement (EBm) obtained the best performance results of the seven evaluated approaches (see Table 2). The statistical analysis performed indicated that EBm outperformed the rest of the approaches in terms of recall, precision, and MCC. Overall, these results confirm that the use of EBm compared with the rest of the approaches has an actual impact. In other words, EBm obtained the best results in recall, which means that the model fragment proposed as the solution has more relevant elements

Table 4. The $\hat{A}_{12}$ statistic for each pair of algorithms and each bug in BSH and CAF

			$\hat{A}_{12}$ for algorithms			$\hat{A}_{12}$ for bugs					
						Recall		Precision		MCC	
			Recall	Precision	MCC	N	Mean± (σ)	N	Mean± (σ)	N	Mean± (σ)
BSH	RQ1	EBRO vs. BLiMEA	0.60701	0.63039	0.67056	32	0.79 ± 0.05	21	0.74 ± 0.07	27	0.83 ± 0.10
		EBRO vs. EB	0.43426	0.39408	0.39007	37	0.07 ± 0.03	37	0.05 ± 0.01	36	0.09 ± 0.06
		EBRO vs. CoEB	0.78415	0.73557	0.87655	37	0.94 ± 0.03	18	0.73 ± 0.06	12	0.65 ± 0.15
		EBRO vs. HC	1	0.98612	1	37	1 ± 0	37	0.95 ± 0.03	37	1 ± 0
		EBRO vs. RS	1	1	1	37	1 ± 0	37	1 ± 0	37	1 ± 0
		BLiMEA vs. EB	0.35391	0.30022	0.25274	37	0.09 ± 0.03	37	0.18 ± 0.10	37	0.01 ± 0.01
		BLiMEA vs. CoEB	0.67458	0.64025	0.73192	37	0.91 ± 0.05	0	0.49 ± 0.07	21	0.72 ± 0.13
		BLiMEA vs. HC	1	0.93608	1	37	1 ± 0	37	1 ± 0	37	1 ± 0
		BLiMEA vs. RS	1	1	1	37	1 ± 0	37	1 ± 0	37	1 ± 0
		EB vs. CoEB	0.78561	0.78524	0.89299	37	0.93 ± 0.04	37	0.92 ± 0.04	37	0.96 ± 0.04
		EB vs. HC	1	0.99014	1	37	1 ± 0	37	1 ± 0	37	1 ± 0
		EB vs. RS	1	1	1	37	1 ± 0	37	1 ± 0	37	1 ± 0
		CoEB vs. HC	0.99781	0.80497	0.98612	37	0.95 ± 0.03	37	0.93 ± 0.05	37	0.94 ± 0.03
		CoEB vs. RS	1	0.94302	1	37	1 ± 0	37	0.94 ± 0.03	37	1 ± 0
		HC vs. RS	0.78634	0.70234	0.78342	0	0.49 ± 0.06	37	0.89 ± 0.07	31	0.78 ± 0.30
	RQ2	EB vs. EmB	0.75310	0.81191	0.90212	37	0.92 ± 0.05	37	0.93 ± 0.04	37	0.92 ± 0.04
		EB vs. EBm	0.33893	0.33747	0.26881	37	0.0.7 ± 0.03	37	0.13 ± 0.07	37	0.07 ± 0.04
		EB vs. EmBm	0.72243	0.71439	0.79036	37	0.99 ± 0.01	37	0.93 ± 0.04	34	0.90 ± 0.08
		EmB vs. EBm	0.10665	0.11029	0.02776	37	0.08 ± 0.04	37	0.09 ± 0.04	37	0.08 ± 0.04
		EmB vs. EmBm	0.44047	0.35865	0.35208	37	0.19 ± 0.09	37	0.08 ± 0.04	37	0.06 ± 0.04
		EBm vs. EmBm	0.88605	0.80716	0.92988	37	0.93 ± 0.04	37	0.92 ± 0.04	37	0.94 ± 0.05
		CoEB vs. CoEBm	0.49744	0.24251	0.28780	37	0.81 ± 0.10	37	0.08 ± 0.04	4	0.50 ± 0.14
		HC vs. HCm	0.39372	0.38203	0.32031	37	0.10 ± 0.06	35	0.16 ± 0.06	37	0.01 ± 0.03
CAF	RQ1	EBRO vs. BLiMEA	0.45362	0.59094	0.53871	56	0.99 ± 0.01	52	0.82 ± 0.05	53	0.92 ± 0.11
		EBRO vs. EB	0.40248	0.30095	0.30606	56	0.10 ± 0.04	56	0.08 ± 0.04	56	0.07 ± 0.04
		EBRO vs. CoEB	0.68590	0.65888	0.78086	56	0.91 ± 0.04	56	0.93 ± 0.05	56	0.99 ± 0.01
		EBRO vs. HC	1	0.98466	1	56	1 ± 0	56	0.95 ± 0.03	56	1 ± 0
		EBRO vs. RS	1	1	1	56	1 ± 0	56	1 ± 0	56	1 ± 0
		BLiMEA vs. EB	0.48868	0.23667	0.29182	56	0.08 ± 0.04	56	0.08 ± 0.04	56	0.08 ± 0.05
		BLiMEA vs. CoEB	0.71877	0.58145	0.71585	56	0.93 ± 0.05	56	0.92 ± 0.04	49	0.90 ± 0.14
		BLiMEA vs. HC	1	0.94083	1	56	1 ± 0	56	0.93 ± 0.05	56	1 ± 0
		BLiMEA vs. RS	1	1	1	56	1 ± 0	56	1 ± 0	56	1 ± 0
		EB vs. CoEB	0.77137	0.80862	0.89993	56	0.92 ± 0.04	56	0.92 ± 0.05	56	0.93 ± 0.04
		EB vs. HC	1	1	1	56	1 ± 0	56	1 ± 0	56	1 ± 0
		EB vs. RS	1	1	1	56	1 ± 0	56	1 ± 0	56	1 ± 0
		CoEB vs. HC	1	0.85026	0.99379	56	1 ± 0	56	0.93 ± 0.04	56	0.93 ± 0.05
		CoEB vs. RS	1	0.95617	1	56	1 ± 0	56	0.92 ± 0.05	56	1 ± 0
		HC vs. RS	0.68517	0.74507	0.79474	56	0.93 ± 0.04	56	0.93 ± 0.04	56	0.99 ± 0.01
	RQ2	EB vs. EmB	0.74434	0.78744	0.84076	56	0.92 ± 0.04	56	0.92 ± 0.04	56	0.93 ± 0.04
		EB vs. EBm	0.23156	0.21622	0.17714	2	0.49 ± 0.09	5	0.49 ± 0.08	3	0.50 ± 0.08
		EB vs. EmBm	0.71147	0.74288	0.80972	56	0.90 ± 0.04	56	0.92 ± 0.05	56	0.92 ± 0.04
		EmB vs. EBm	0.12783	0.06720	0.04894	56	0.08 ± 0.04	56	0.05 ± 0.03	56	0.08 ± 0.04
		EmB vs. EmBm	0.38860	0.40102	0.37619	56	0.02 ± 0.01	56	0.08 ± 0.05	52	0.08 ± 0.11
		EBm vs. EmBm	0.85756	0.89774	0.94814	56	0.90 ± 0.03	56	0.93 ± 0.04	56	0.92 ± 0.05
		CoEB vs. CoEBm	0.59386	0.33674	0.44595	56	0.08 ± 0.04	56	0.07 ± 0.04	56	0.07 ± 0.05
		HC vs. HCm	0.27100	0.39445	0.26114	55	0.17 ± 0.05	56	0.07 ± 0.04	51	0.06 ± 0.13

for the bug that must be located than the model fragments proposed by the rest of the approaches. In the same way, EBm obtained the best results in precision, which means that the model fragment proposed as the solution has fewer non-relevant elements for the bug that must be located than the model fragments proposed by the rest of the approaches.

6.5 Discussion

Some might think that the more human knowledge is involved in the search for bugs, the better the results will be. This should be true, especially when those humans have as much knowledge of the domain and the software as developers do. When we started this evaluation, we also thought that the more human knowledge through manual refinement, the better the results would be. After the evaluation, we have realized that in order to involve humans, we must do so at the right time. If we do not, the results can even get worse.

In the case of EmB and EmBm, we apply BLiMEA on the manually refined solution obtained in EBRo. When humans try to refine the solutions provided by the EBRo approach, the results get worse by about a 10%. The output of the EBRo approach is a reconfiguration sequence that might trigger the target bug. In the mind of humans, some solutions are invalid sequences of reconfigurations that are not going to take place since the necessary context changes should never occur. Hence, software engineers try to modify these sequences (adding or deleting reconfigurations) in order to obtain a sequence that should take place. However, induction hobs and trains are sold all over the world and sometimes they are used in unforeseen ways. This can cause extremely unlikely sequences of reconfigurations to occur. We realize that these unlikely reconfiguration sequences must be taken into account since these reconfigurations (which the software engineers believe are impossible) are actual sources of bugs.

In the case of EBm, we apply BLiMEA on the solution obtained in EBRo and the solution is then manually refined. When humans refine the solutions, the results improve. The output of the BLiMEA approach is a model fragment that might contain the bug. Humans can add or remove model elements to the model fragment. There are some model elements that have little or no text, e.g., the channels that connect the inverters and the inductors with the power manager. In our approach, even though these elements without enough text do not obtain a good assessment for measuring the textual similarity, humans can see these model elements and decide whether or not they should be present in the solution. Humans can better evaluate elements that the algorithm cannot measure.

As a result of these observations, we can conclude that we should stop considering the involvement of humans as something binary (good or bad). We should think that humans have different types of knowledge that they can contribute to algorithms. Humans, especially those who are adept at locating bugs in software systems, are often a scarce resource in the industry. Therefore, we should evaluate when and what knowledge is good for each problem and thereby avoid wasting resources.

We started this work stating that we wanted to evaluate how to apply the existing approaches in order to mitigate the effect of starting the localization in the wrong place. The results obtained and the statistical analysis show that there is no statistical significance in the results when we perform the bug localization with EBRo or BLiMEA separately.

When models are used to develop software, as in Model-Driven Development (MDD), the main development artefact is the product model. In MDD contexts, engineers specify the system to be built with the models and then obtain the source code through model transformations. When models are the main development artefact, bug localization is performed on product models. This is the case of our industrial partners, which use a DSL to specify the firmware of the induction hobs and the software to control the trains. The C++ code that controls their products is obtained

with a transformation from model to code. If the bug is part of the initial model used by EBRo, the approach could generate a reconfiguration sequence in which the reconfigurations do not manipulate the elements of the initial model that are relevant to the bug. However, if a bug is in the source code and it is not represented in the models, it might not be found with these model-based feature localization approaches. There are also MDD contexts in which the models are interpreted. An example of interpreted models is the case of Kromaia, a commercial video game where all of its elements are specified with a DSL that is interpreted at runtime [10]. In these cases, no code is generated from the models, so bug localization is performed on a product model.

We realized that the approaches EBRo and BLiMEA can be combined so that the results are improved somewhat without worsening. In other words, the combination of EBRo and BLiMEA improves the results without running the risk of doing the opposite, namely worsening the process. EBRo obtains around 0.74 in performance, while EBRo plus BLiMEA (EB) reaches 0.77 in performance. Since statistical analysis also shows that the results are not significant, the current tendency to use one approach or another does not make sense. However, the effort required to perform the combination is very small. We can easily obtain a model from the output of the EBRo approach that can be used as input in the BLiMEA approach. This combination ensures that the solution is assessed taking into account the run-time and the design-time information. EBRo provides the textual similarity from runtime, and BLiMEA provides the textual similarity and the defect localization principle from design time. Thus, since the effort is small and the results improve slightly, we can combine our approaches, providing the advantage that the software engineers do not have to choose which approach to use in order to perform bug localization.

In summary, the conclusions obtained from this evaluation are the following: there is no statistical significance between using one of our approaches or another; the EBRo and BLiMEA approaches can be combined without worsening the results; and the introduction of manual refinement pays off if it is used at the right time.

6.6 Threats to Validity

In this section, we present some of the threats to validity. We follow the guidelines suggested by De Oliveira et. al [16] to identify those that are applicable to this work.

Conclusion validity threats: To avoid the *not accounting for random variation* threat, we considered 30 independent runs for each bug with each algorithm. In this paper, we employed standard statistical analysis following accepted guidelines [5] to avoid the *lack of formal hypothesis and statistical tests* threat. To address the *lack of good descriptive analysis* threat, we have used precision, recall, F-measure, and MCC metrics to analyze the confusion matrix obtained from the experiments; however, other metrics could be applied. In addition, some works argue that the use of the Vargha and Delaney $\hat{A}_{12}$ metric can be misrepresentative [38] and that the data should be pre-transformed before applying the metrics. We did not find any use case for data pre-transformation that applies to our case study.

Internal validity threats: In this paper, we used standard values for the algorithms to avoid the *poor parameter settings* threat. These values have been tested in similar algorithms for feature localization [20]. The evaluation of this paper was applied to an industrial case study from our partner, BSH; hence, the *lack of real problem instances* threat has been addressed. To avoid the threat of *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. In addition, a common pitfall of involving humans in the studies is a "training effect" wherein the participants do significantly worse at the beginning of the study due to unfamiliarity with the task. This can potentially invalidate results drawn from the

affected data. To address this concern, we select the team of our industrial partner that has work experience performing bug localization in their model-based family.

Construct validity threats: To address the *lack of assessing the validity of cost measures* threat, we performed a fair comparison among EBRo, BLiMEA, CoEB, HC, the combinations, and the random approaches by generating the same number of solutions and using the same number of fitness evaluations.

External validity threats: The *lack of a clear object selection strategy* and *lack of evaluations for instances of growing size and complexity* threats are addressed by using an industrial case study from our partner, BSH. Our instances are collected from real-world problems. On the other hand, the limited number of participants calls for further studies on the subject. Specific high-complexity management software engineering tasks, such as bug localization, may require highly specialized expertise, which limits the humans that can be involved in the evaluation.

7 RELATED WORK

This section presents related works. It is divided into three subsections. The first two subsections take into account the topics covered in this paper: bug localization approaches and studies on bug localization. The third subsection discusses the overlap with our previous works.

7.1 Bug Localization approaches

In recent years, many bug localization approaches have been proposed [53]. These approaches are usually IR-based approaches, and some of them add the defect localization principle. Since our bug localization approach applies these techniques, in this section, we review some relevant works in the literature.

Troya et al. [48] present an approach to apply Spectrum-Based Fault Localization (SBFL) for locating the faulty rules in model transformations. Their approach takes advantage of the information recovered after model transformation runs. The inputs of their approach are a model transformation, a set of assertions, and a set of source models. Their approach indicates the violated assertions and uses the information of the model transformation coverage to rank the transformation rules according to their suspiciousness of containing a bug.

Lam et al. [31] present an approach that uses a deep neural network (DNN) in combination with an information retrieval technique, rVSM. The rVSM technique selects the feature based on the textual similarity between bug reports and source files. DNN is used to learn the terms in bug reports or to relate them to the potentially different code tokens and terms in source files.

Sánchez-Cuadrado et al. [43] combine static analysis and constraint solving to discover errors in ATL transformations. They developed a tool that uses static analysis to detect problems based on the typing information and generates witness models using OCL path conditions and constraint solving. In their subsequent works, they present an approach that proposes suitable quick fixes for ATL transformation errors [44]. Their approach performs speculative analysis to provide information on the impact of the application of each applicable quick fix and generates a dynamic quick fix rank. In addition, they constructed a static ranking empirically through the automated application of quick fixes on transformations.

Burgueño et al. [13] present a static approach to trace errors in model transformations, taking as input an ATL model transformation and a set of constraints that specify its expected behavior. Their approach automatically extracts the footprints of both artefacts and compares transformation rules and constraints one by one, obtaining the overlap of common footprints. The output is three matching tables that can be used by software engineers to trace the rules that might be the cause of broken constraints due to faulty behavior.

Zamani et al. [54] include the defect localization principle. They propose 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. Sisman and Kak [46] include the defect localization principle in their approach. They utilize time decay in weighting the files in a probabilistic information retrieval model.

Wimmer et al. [52] propose a model-based debugger for Query/View/Transformation (QVT) Relations. They employ TRansformations On Petri nets In Color (TROPIC). Their debugging environment helps programmers to observe facts, find the origins of errors, and correct them in the debugger itself. Similar to Wimmer et al., Schoenboeck et al. [45] propose Transformation Nets (TNs), which is a DSL on top of Colored Petri Nets (CPNs), for developing, executing, and debugging model transformations. Their approach allows debugging and enables the verification of model transformations.

The above approaches that take into account models only consider model transformations as the source of the bug. The rest of the above works only take into account the source code as the artefact that represents the bug. When models are used for code generation, addressing bugs at the model level should not be neglected.

Sometimes, with design-time approaches, we include extra information that is not relevant for a bug that occurs at runtime. To exclude this extra information, we need approaches that take into account the run-time behavior. Some of these approaches are the following.

Wang and Lo [51] present AmaLgam+, which is a method for locating relevant buggy files that puts together five sources of information: version history, similar reports, structure, stack traces, and reporter information. Le et al. [33] and Hoang et al. [27] combine information retrieval and spectrum-based techniques. In [27, 33], they present two approaches that utilize multi-modal information from both bug reports and program spectra to localize bugs.

Mao et al. [36] 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. Gong et al. [23] propose an interactive localization technique called TALK. This approach incorporates programmers' feedback into spectrum-based fault localization techniques. When programmers receive the ranking of program elements that can cause the bug, they can judge the correctness of each element and provide this information as feedback to reorder the ranking.

Similar to the previous design-time approaches, none of the approaches presented above take into account the models or their reconfigurations at runtime as the source of the bugs. In many systems with models at runtime, the models experience reconfigurations at runtime due to context changes, with these reconfigurations being a source of bugs.

7.2 Studies on Bug Localization

There are many research efforts that evaluate bug localization techniques and the combination of these techniques. Chaleshtare et al. [14] present a report on an empirical study of two state-of-the-art fault localization techniques. They focus on spectrum-based and mutation-based methods, considering different suspiciousness formulas to limit their effect on fault localization accuracy and two different test suites containing small- and large-scale projects to test all of the aspects of the methods.

Jiang et al. [29] perform an empirical study on the combination of spectrum-based fault localization (SBFL) and statistical debugging (SD) families, both of which localize faults by collecting statistical information at runtime. They explore four types of variations: different predicates, different risk evaluation formulas, different granularities of data collection, and different methods of combining suspiciousness scores. Their findings are: most of the effectiveness of the combined

approach is contributed by a simple type of predicate (branch conditions); the risk evaluation formulas of SBFL significantly outperform those of SD; fine-grained data collection significantly outperforms coarse-grained data collection with a little extra execution overhead; and a linear combination of SBFL and SD predicates outperforms both individual approaches.

Zou et al. [56] investigate the performance of a wide range of fault localization techniques (including eleven techniques from seven families) on 357 real-world faults. They evaluated the effectiveness of each standalone fault localization technique. They applied a learning-to-rank model to combine these fault localization techniques. The combined techniques significantly outperform any standalone technique.

Pearson et al. [40] evaluated spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL) techniques on both artificial and real-world faults to determine whether the previous findings for artificial faults still hold for real-world faults. They identify several cases where results for artificial faults are different from those for real-world faults, indicating that experimenting on real-world faults is important. In other words, results from the Siemens test suite are not characteristic of real-world faults.

All of these works focus on existing techniques and their combinations applied to program code. In this evaluation, we evaluate four approaches that combine different techniques applied to models. In addition, in contrast to previous works, we also take into account that software engineers (humans) can refine the results obtained. Other research works on bug localization that involve humans are focused on studies or surveys [21, 50]. Although our work involves few humans, they are from industry, which is a very difficult profile to find, and they spent approximately one-month doing manual refining work for the evaluation of this paper.

7.3 Our previous works

In this paper, we evaluate how to apply our previous model-based approaches [3, 4] in order to mitigate the effect of starting the localization in the wrong place. At first glance, this paper looks like a continuation of our previous works regarding bug localization because we use our previous bug localization approaches in the evaluation. However, this paper is not just an extension of our previous work. This paper takes another step forward in improving bug localization in model-based systems, evaluating how to combine the approaches if the software engineers do not know the artefact that causes the bug. In fact, this new paper is inspired by our previous experience in real industrial environments. With this paper, we can confirm the good results that our previous bug localization approaches obtained, which were previously evaluated using the same case studies from two different domains. The new insight extracted with this research work is that the combination of our approaches is possible and improves the results.

Moreover, the paper further explores how involving humans affect bug localization results. We take into account that software engineers can refine the results and we provide them with the ability to modify the solution obtained in different stages of the bug localization process. However, we have realized that humans must get involved at the right time; otherwise, the results may not improve but rather get worse.

8 CONCLUSION

Bug localization is a significant maintenance activity. However, even though companies that have adopted the MDE paradigm have the advantage of working at a high level of abstraction, they have the disadvantage of a lack of approaches to perform bug localization at the model level. In addition, in an MDE context, a bug can be related to different MDE elements.

In this paper, we have evaluated how to apply our previously developed approaches, EBRo and BLiMEA in order to mitigate the effect of starting the localization in the wrong place. In our

evaluation, we have compared EBRo, BLiMEA, CoEB, HC, the combination of EBRo and BLiMEA. In addition, we have taken into account that software engineers (humans) can manually refine the results obtained by each of the approaches.

We evaluated our approaches in two industrial case studies, BSH and CAF. We determined which approach produces the best results in terms of recall, precision, the F-measure, and MCC. To do this, we applied the approaches to two model-based product families of our industrial partners. The results show that the combination of the EBRo approach and the BLiMEA approach plus manual refinement outperforms the rest of the approaches. The results in terms of recall, precision, and MCC, on average, are 0.89, 0.80, and 0.82, respectively. The statistical analysis of the results provides evidence of their significance.

In summary, the conclusions obtained from this evaluation are: the introduction of manual refinement pays off if it is used at the right time, and artefact-dependent approaches are not necessarily the only path for bug localization in MDE. This is the first paper that studies manual refinement in bug localization in the context of MDE. We hope that our results will motivate and help more researchers to address the bug localization task in MDE. Despite more than 20 years of MDE development, bug localization in MDE has not yet received enough attention.

ACKNOWLEDGMENTS

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 ALPS (RTI2018-096411-B-I00).

REFERENCES

[1] Michael Affenzeller, Stephan M. Winkler, Stefan Wagner, and Andreas Beham. 2009. *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications.* CRC Press, United Kingdom. http://www.crcpress.com/product/isbn/9781584886297

[2] Lorena Arcega. 2019. *Feature and Bug Localization on Model-based Systems at Design time and Runtime.* Ph.D. Dissertation. University of Oslo, Norway.

[3] Lorena Arcega, Jaime Font, and Carlos Cetina. 2018. Evolutionary Algorithm for Bug Localization in the Reconfigurations of Models at Runtime. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18).* Association for Computing Machinery, New York, NY, USA, 90–100. https://doi.org/10.1145/3239372.3239392

[4] Lorena Arcega, Jaime Font, Øystein Haugen, and Carlos Cetina. 2019. An approach for bug localization in models using two levels: model and metamodel. *Software and Systems Modeling* 18, 6 (2019), 3551–3576. https://doi.org/10.1007/s10270-019-00727-y

[5] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219 – 250. https://doi.org/10.1002/stvr.1486

[6] A. Arcuri and G. Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

[7] James E. Baker. 1987. Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application* (Cambridge, Massachusetts, USA). L. Erlbaum Associates Inc., USA, 14–21. http://dl.acm.org/citation.cfm?id=42512.42515

[8] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@ run. time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* 18, 5 (2019), 3049–3082.

[9] Gordon Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ Run.Time. *Computer* 42, 10 (Oct. 2009), 22–27. https://doi.org/10.1109/MC.2009.326

[10] Daniel Blasco, Jaime Font, Mar Zamorano, and Carlos Cetina. 2021. An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering. *Journal of Systems and Software* 171 (2021), 110804. https://doi.org/10.1016/j.jss.2020.110804

[11] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. 2017. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *PLOS ONE* 12, 6 (06 2017), 1–17. https://doi.org/10.1371/journal.pone.0177678

[12] Mohamed Boussaa, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. 2013. Competitive Coevolutionary Code-Smells Detection. In *Search Based Software Engineering*, Günther Ruhe and Yuanyuan Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65.

[13] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. 2015. Static Fault Localization in Model Transformations. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 490–506. https://doi.org/10.1109/TSE.2014.2375201

[14] Nazanin Bayati Chaleshtari and Saeed Parsa. 2020. A Comparison Study of Two Well-known Fault Localization Methods. In *2020 25th International Computer Conference, Computer Society of Iran (CSICC)*. IEEE, Tehran, Iran, 1–6.

[15] W. J Conover. 1999. *Practical Nonparametric Statistics, 3rd Edition*. Wiley, USA.

[16] Márcio de Oliveira Barros and Arilo Cláudio Dias-Neto. 2011. *0006/2011-Threats to Validity in Search-based Software Engineering Empirical Studies*. Technical Report 1. Universidade Federal do Estado do Rio de Janeiro.

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[18] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407. https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9

[19] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2011. Feature Location in Source Code: A Taxonomy and Survey. In *Journal of Software Maintenance and Evolution: Research and Practice*. Wiley, USA, 53–95.

[20] Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. 2018. Achieving Feature Location in Families of Models Through the Use of Search-Based Software Engineering. *IEEE Trans. Evolutionary Computation* 22, 3 (2018), 363–377. https://doi.org/10.1109/TEVC.2017.2751100

[21] Zachary P. Fry and Westley Weimer. 2010. A Human Study of Fault Localization Accuracy. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609691

[22] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. 2010. 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 (2010), 2044 – 2064. https://doi.org/10.1016/j.ins.2009.12.010 Special Issue on Intelligent Distributed Information Systems.

[23] Liang Gong, Hongyu Zhang, Lingxiao Jiang, and David Lo. 2012. Interactive Fault Localization Leveraging Simple User Feedback. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM) (ICSM '12)*. IEEE Computer Society, USA, 67–76. https://doi.org/10.1109/ICSM.2012.6405255

[24] R. J. Grissom and J. J. Kim. 2005. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, New Jersey, United States.

[25] Ahmed E. Hassan and Richard C. Holt. 2005. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, USA, 263–272. https://doi.org/10.1109/ICSM.2005.91

[26] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, G?ran K. Olsen, and Andreas Svendsen. 2008. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 2008 12th International Software Product Line Conference (SPLC '08)*. IEEE Computer Society, Washington, DC, USA, 139–148. https://doi.org/10.1109/SPLC.2008.25

[27] Thong Hoang, Richard Jayadi Oentaryo, Tien-Duy B. Le, and David Lo. 2019. Network-Clustered Multi-Modal Bug Localization. *IEEE Transactions on Software Engineering* 45, 10 (2019), 1002–1023. https://doi.org/10.1109/TSE.2018.2810892

[28] H. Ishibuchi, Y. Nojima, and Tsutomu Doi. 2006. Comparison between Single-Objective and Multi-Objective Genetic Algorithms: Performance Comparison and Performance Measures. In *2006 IEEE International Conference on Evolutionary Computation*. 1143–1150. https://doi.org/10.1109/CEC.2006.1688438

[29] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, USA, 502–514. https://doi.org/10.1109/ASE.2019.00054

[30] D. Kim, Y. Tao, S. Kim, and A. Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Transactions on Software Engineering* 39, 11 (Nov 2013), 1597–1610. https://doi.org/10.1109/TSE.2013.24

[31] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, USA, 218–229. https://doi.org/10.1109/ICPC.2017.24

[32] Thomas K Landauer, Peter W. Foltz, and Darrell Laham. 1998. An introduction to latent semantic analysis. *Discourse Processes* 25, 2-3 (1998), 259–284. https://doi.org/10.1080/01638539809545028 arXiv:http://dx.doi.org/10.1080/01638539809545028

[33] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[34] Sean Luke. 2012. *Essentials of Metaheuristics* (second edition ed.).

[35] Usman Mansoor, Marouane Kessentini, Philip Langer, Manuel Wimmer, Slim Bechikh, and Kalyanmoy Deb. 2015. MOMM: Multi-objective model merging. *Journal of Systems and Software* 103 (2015), 423 – 439. https://doi.org/10.1016/j.jss.2014.11.043

[36] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based Statistical Fault Localization. *J. Syst. Softw.* 89 (March 2014), 51–62. https://doi.org/10.1016/j.jss.2013.08.031

[37] L. Miguel Antonio and C. A. Coello Coello. 2018. Coevolutionary Multiobjective Evolutionary Algorithms: Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation* 22, 6 (2018), 851–865. https://doi.org/10.1109/TEVC.2017.2767023

[38] Geoffrey Neumann, Mark Harman, and Simon Poulding. 2015. *Transformed Vargha-Delaney Effect Size*. Springer International Publishing, Cham, 318–324. http://dx.doi.org/10.1007/978-3-319-22183-0_29

[39] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. 2016. 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. 314–325. https://doi.org/10.1109/SANER.2016.97

[40] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[41] D. M. W. Powers. 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[42] Christopher D. Rosin and Richard K. Belew. 1997. New Methods for Competitive Coevolution. *Evol. Comput.* 5, 1 (March 1997), 1–29. https://doi.org/10.1162/evco.1997.5.1.1

[43] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. 2017. Static Analysis of Model Transformations. *IEEE Transactions on Software Engineering* 43, 9 (2017), 868–897. https://doi.org/10.1109/TSE.2016.2635137

[44] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. 2018. Quick Fixing ATL Transformations with Speculative Analysis. *Softw. Syst. Model.* 17, 3 (July 2018), 779–813. https://doi.org/10.1007/s10270-016-0541-1

[45] Johannes Schoenboeck, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2009. Catch Me If You Can – Debugging Support for Model Transformations. In *Proceedings of the 2009 International Conference on Models in Software Engineering* (Denver, CO) *(MODELS'09)*. Springer-Verlag, Berlin, Heidelberg, 5–20. https://doi.org/10.1007/978-3-642-12261-3_2

[46] Bunyamin Sisman and Avinash C. Kak. 2012. Incorporating Version Histories in Information Retrieval Based Bug Localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (Zurich, Switzerland) *(MSR '12)*. IEEE Press, USA, 50–59.

[47] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and G?ran K. Olsen. 2010. 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* (Jeju Island, South Korea) *(SPLC'10)*. Springer-Verlag, Berlin, Heidelberg, 106–120. http://dl.acm.org/citation.cfm?id=1885639.1885650

[48] Javier Troya, Sergio Segura, Jose Antonio Parejo, and Antonio Ruiz-Cortés. 2018. Spectrum-Based Fault Localization in Model Transformations. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 13 (Sept. 2018), 50 pages. https://doi.org/10.1145/3241744

[49] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 arXiv:http://jeb.sagepub.com/content/25/2/101.full.pdf+html

[50] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-Based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2771783.2771797

[51] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942. https://doi.org/10.1002/smr.1801 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1801

[52] Manuel Wimmer, Gerti Kappel, Johannes Schoenboeck, Angelika Kusel, Werner Retschitzegger, and Wieland Schwinger. 2009. A Petri Net Based Debugging Environment for QVT Relations. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, USA, 3–14. https://doi.org/10.1109/ASE.2009.99

[53] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug 2016), 707–740.

[54] Sima Zamani, Sai Peck Lee, Ramin Shokripour, and John Anvik. 2014. A noun-based approach to feature location using time-aware term-weighting. *Information and Software Technology* 56, 8 (2014), 991 – 1011. https://doi.org/10.1016/j.infsof.2014.03.007

[55] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 563–572. http://dl.acm.org/citation.cfm?id=998675.999460

[56] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* - (2019), 1–1.