A Fine-Grained Requirement Traceability Evolutionary Algorithm: Kromaia, a Commercial Video Game Case Study

Daniel Blasco*, Carlos Cetina, Óscar Pastor

Universidad San Jorge. SVIT Research Group
Autovía A-23 Zaragoza-Huesca Km.299, 50830, Zaragoza, Spain

Abstract

Context: Commercial video games usually feature an extensive source code and requirements that are related to code lines from multiple methods. Traceability is vital in terms of maintenance and content update, so it is necessary to explore such search spaces properly.

Objective: This work presents and evaluates CODFREL (Code Fragment-based Requirement Location), our approach to fine-grained requirement traceability, which lies in an evolutionary algorithm and includes encoding and genetic operators to manipulate code fragments that are built from source code lines. We compare it with a baseline approach (Regular-LSI) by configuring both approaches with different granularities (code lines / complete methods).

Method: We evaluated our approach and Regular-LSI in the Kromaia video game case study, which is a commercial video game released on PC and PlayStation 4. The approaches are configured with method and code line granularity and work on 20 requirements that are provided by the development company. Our approach and Regular-LSI calculate similarities between requirements and code fragments or methods to propose possible solutions and, in the case of CODFREL, to guide the evolutionary algorithm.

Results: The results, which compare code line and method granularity configurations of CODFREL with different granularity configurations of Regular-LSI, show that our approach outperforms Regular-LSI in precision and recall, with values that are 26 and 8 times better, respectively, even though it does not achieve the optimal solutions. We make an open-source implementation of CODFREL available.

Conclusions: Since our approach takes into consideration key issues like the source code size in commercial video games and the requirement dispersion, it provides better starting points than Regular-LSI in the search for solution candidates for the requirements. However, the results and the influence of domain-specific language on them show that more explicit knowledge is required to improve such results.

Keywords: Requirement, Traceability, Evolutionary Computation, Video Game, Source Code

*Corresponding author. Tel.: +34 976060100
Email addresses: `dblasco@usj.es` (Daniel Blasco), `ccetina@usj.es` (Carlos Cetina), `opastor@dsic.upv.es` (Óscar Pastor)

1. Introduction

Traceability has been shown by researchers to have a significant impact on successful software engineering projects [1]. This has encouraged reliability and maintainability improvement efforts to trace and verify critical, non-reliable sections in software systems [2]. Traceability Link Recovery (TLR) has been studied by software engineers for a considerable number of years [3], [4], and low defect rates in software products are associated to more complete Traceability [5].

In the video game industry, the titles classified as AAA are those produced and distributed by a mid-sized or major publisher and are typically more complex than regular games and have high development and marketing budgets. Due to the nature of real-time physics simulation, high numbers of interacting entities, and long chains of events that branch and have a significant impact on the course of the game, AAA video games feature many requirements that involve more than one complete method. Current traceability approaches [6] evaluate methods in the source code of a software product as atomic units. The purpose of approaches of this type is to decide which method or method set is the best candidate for a given requirement, conceiving requirements as functionalities that are likely to be mainly defined by one or various complete methods in the source code of a software product. These approaches are not suitable for AAA video games featuring dispersed requirements.

Our approach, Source Code Fragment-based Requirement Location (CODFREL), generates possible solutions that are more flexible than complete methods. Our approach relies on an evolutionary algorithm which:

- searches for flexible solution candidates, represented by **code fragments**, i.e., sets of code lines belonging to the source code that are not necessarily contiguous or included in a single method.
- explores the search space through **genetic operations** involving mutation and fusion. The search process is guided by similitude evaluation between the terms present in a code fragment and those specified by the requirement; this similitude is measured through **Latent Semantic Indexing** (LSI) [7].

Our evaluation compares our approach to Regular-LSI. Regular-LSI uses LSI, but it does not use code fragments or an evolutionary algorithm. We compare our CODFREL approach to Regular-LSI using requirements from Kromaia, a commercial video game case study. Kromaia is a physics-based space simulation video game that was developed by Kraken Empire (www.krakenempire.com) and published by Rising Star Games (www.risingstargames.com). This is a title that has been released worldwide both digitally and physically, translated to eight languages, and ported from PC to PlayStation 4.

In order to evaluate our approach and Regular-LSI, we have configured different versions of the two approaches: our CODFREL approach was studied using code fragments and complete method granularity; and Regular-LSI was configured with three different cut-off strategies to search for the best complete method, a set containing the 10 best complete methods, and a set containing the 10 best code lines, considering each code line as a method. The results obtained show that, in comparison to Regular-LSI, our approach provides better solutions, both in terms of precision and recall, for requirements that are dispersed within the source code. Precision, recall and F-measure are information retrieval metrics that are widely used [8]. These results show that, using the different configurations mentioned, our CODFREL approach outperforms Regular-LSI in precision, recall, and F-measure, with average values of 57% and 28% for precision, around 27% for recall, and 29% and 21% for F-measure. Regular-LSI obtained average values of 4%, 0.7%, and 0.1% for precision, for the different configurations used. For recall, the values were 0.5%, 0.5%, and 9%. F-measure reached values of 0.9%, 0.6%, and 0.2%.

Both our approach and Regular-LSI output a solution that has to be manually refined to obtain all of the code that is relevant to the requirement. The solutions generated by our approach are better starting points in comparison to Regular-LSI. However, our approach does not obtain perfect solutions (every relevant code line in a requirement). Tacit knowledge, which is not written in the requirements, causes this issue. Therefore, we plan to deal with this matter through reformulations that expand the requirements with descriptions provided by domain experts.

The structure used in the paper is the following: Section 2 describes the motivation for our work. Section 3 presents an overview of our CODFREL approach. Section 4 presents the code fragment encoding. Section 5 describes the operations used in code fragment generation. Section 6 discusses how code fragment suitability is determined. Section 7 deals with the evaluation, comparing our approach to Regular-LSI, and Section 8 discusses the results obtained. Section 9 deals with future work, Section 10 describes the threats to validity, and Section 11 summarizes related works. Finally, Section 12 presents our conclusions.

2. Motivation

Commercial video games, like Kromaia, often behave like real-time simulation applications, which coordinate internal update processes and data structures that render information as audiovisual data. This data is meant to be coherent with internal logics, but it is not necessarily directly related to them in terms of locality.

Apart from various libraries used in the project, Kromaia features a considerably high number of private code lines (over 260,000) resulting from two main blocks, both of which are owned by the development company: a proprietary game engine, and the video game source code (VGSC). The case study focuses on the VGSC, which contains over 145,000 lines of code.

The use of an evolutionary algorithm emerges as a response to a situation in which the solution space is huge and the requirements are dispersed. Taking into account the VGSC, the current traceability approaches [6] (which feature method granularity) are required to evaluate approximately 9000 complete methods. However, code fragments (the granularity in our approach) may feature any code line in the VGSC, so the total number of possible code fragments in the VGSC is over $2^{145,000}$. Our approach, CODFREL, works with code fragments and uses an evolutionary algorithm to search such a large solution space.

3. Overview of the CODFREL Approach

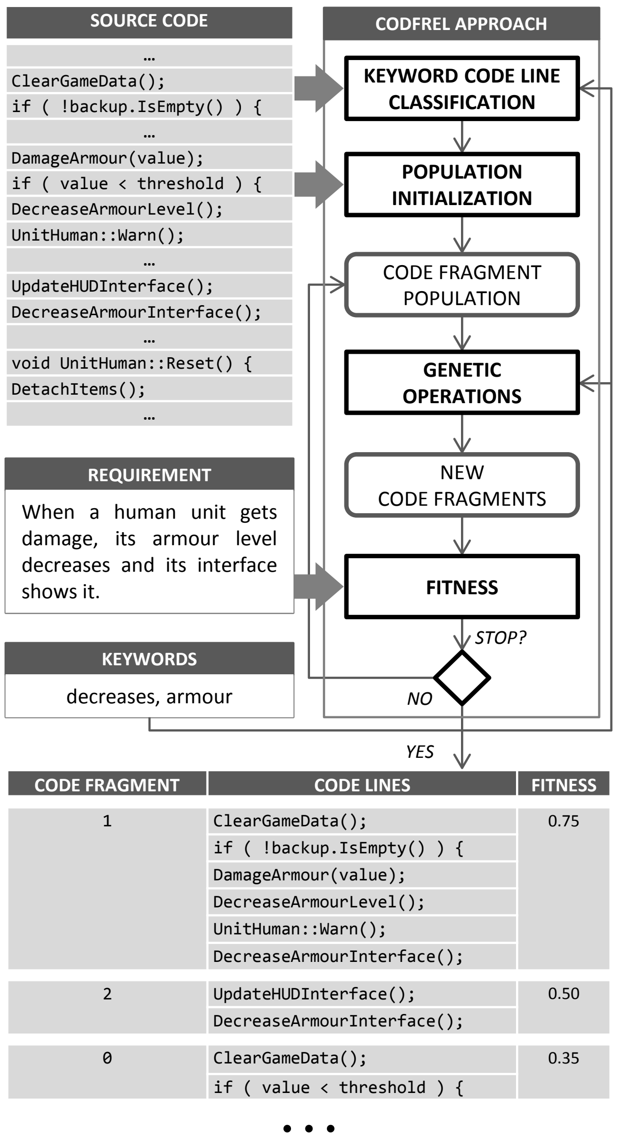
This section presents our CODFREL approach. The approach has a clear goal: to obtain the code fragment from the source code that realizes a requirement that is specified in a natural language. The evolutionary algorithm of CODFREL iterates a code fragment population, which evolves through genetic operations inspired by processes that are present in nature. This evolutionary algorithm is driven by a fitness operation that takes into consideration the requirement. In the end, the output delivered is a ranking, which is a list sorted by a fitness value that features code fragments that might realize the requirement described.

The upper left section in Fig. 1 shows an example of input to our CODFREL approach.

- The **Source Code**. In this paper, we evaluate our approach on the Kromaia VGSC, and we also use it as a running example.
- The **Requirement** (in natural language) to be located in our case study. The requirement is in the Game Design Document created for the video game. The requirement terms may include general vocabulary that is commonly used in the video game industry and, more specifically, terms of the specific video game that are likely to be found in the VGSC.
- In order to minimize the effects of both irrelevant and deceiving terms, our approach asks to select the most relevant terms in the requirement. These terms, the **Keywords**, help our approach to find starting points and provide guidance to the evolutionary algorithm.

The section on the right of Fig. 1 shows the main steps in our approach.

- First, the **Keyword Code Line Classification** step identifies those code lines from the VGSC that feature keywords, which are tagged terms in



```
…
ClearGameData();
if ( !backup.IsEmpty() ) {
…
DamageArmour(value);
if ( value < threshold ) {
DecreaseArmourLevel();
UnitHuman::Warn();
…
UpdateHUDInterface();
DecreaseArmourInterface();
…
void UnitHuman::Reset() {
DetachItems();
…
```

When a human unit gets damage, its armour level decreases and its interface shows it.

decreases, armour

CODE FRAGMENT	CODE LINES	FITNESS
1	`ClearGameData();`	0.75
	`if ( !backup.IsEmpty() ) {`	
	`DamageArmour(value);`	
	`DecreaseArmourLevel();`	
	`UnitHuman::Warn();`	
	`DecreaseArmourInterface();`	
2	`UpdateHUDInterface();`	0.50
	`DecreaseArmourInterface();`	
0	`ClearGameData();`	0.35
	`if ( value < threshold ) {`	

• • •

Figure 1: CODFREL Overview

the requirement. These lines are used in both the initial code fragment population creation step and in those genetic operations involving the expansion of existing code fragments.

- The **Population Initialization** step calculates the starting code fragment population, which is extracted from the VGSC using code lines determined by the keywords. Each initial code fragment is generated as follows: first, a random code line with keyword presence is selected as the starting point. Then, the code fragment is completed by adding a random number of code lines that are placed before and after the starting code line. Random initialization is a common practice in evolutionary computation.

- The **Genetic Operations** step produces a new code fragment generation. This step involves the use of a selection operator that chooses the code fragments that will be the parents of the new generation. This selection, which is done using fitness values, is meant to promote the best code fragments in the population to be parents. Then, new code fragments are produced by mixing code lines from two parents through a fusion operation. This step also introduces modifications in the new code fragments using a guided mutation operation (by adding or removing code lines from the code fragment), which hopefully would make the new code fragments reach fitness values that exceed what their parents achieved.

- The **Fitness** step evaluates the code fragments by assigning a value that depends on the similarity between each code fragment and the requirement. The code fragments that share more terms with the requirement will get the highest values.

The process is over when a code fragment features a fitness value that is greater than a predefined threshold or once a certain time limit is reached. As a result, our CODFREL approach produces a code fragment set, where each code fragment is relevant to the requirement (see the lower section in Fig. 1). The set may be organized as a ranking, ordering the code fragments by similarity to the requirement.

In summary, our CODFREL approach ultimately searches for code fragments that are relevant to the requirement. In order to succeed, the approach creates/iterates a code fragment population and searches within that population using a fitness function that evaluates code fragments by assigning values that depend on the similarity to the requirement defined in natural language.

Sections 4, 5, and 6 show how code fragments are encoded in our CODFREL approach as well as the genetic operations applied and the criteria used by the fitness function to assign different values to code fragments depending on how similar they are to the requirement.

4. Code Fragment Encoding of the CODFREL Approach

The code fragments generated by our approach represent the solutions proposed for the requirement. These solutions need to be encoded, a task that, in evolutionary algorithms, is usually done by storing possible solutions as arrays or strings containing binary values such as 0/1 or true/false.

In our CODFREL approach, where the solutions are the code fragments, the encoding is as follows: each code fragment is a list of code line elements; and a code line element contains indexing information that is relative to the file from which the code line was taken, its local position in that file, and its global position relative to the complete VGSC. Therefore, the encoding used in our approach to represent code fragments is not a fixed length structure, but a variable length set of code lines that is ordered, taking into account the relative positions of each of those lines in the VGSC.

5. Genetic Operations of the CODFREL Approach

Our CODFREL approach generates new code fragments using the existing ones as parents and making use of two genetic operators: fusion and guided mutation. We adapted these operators to function with code fragments, which represent a code line set from the VGSC.

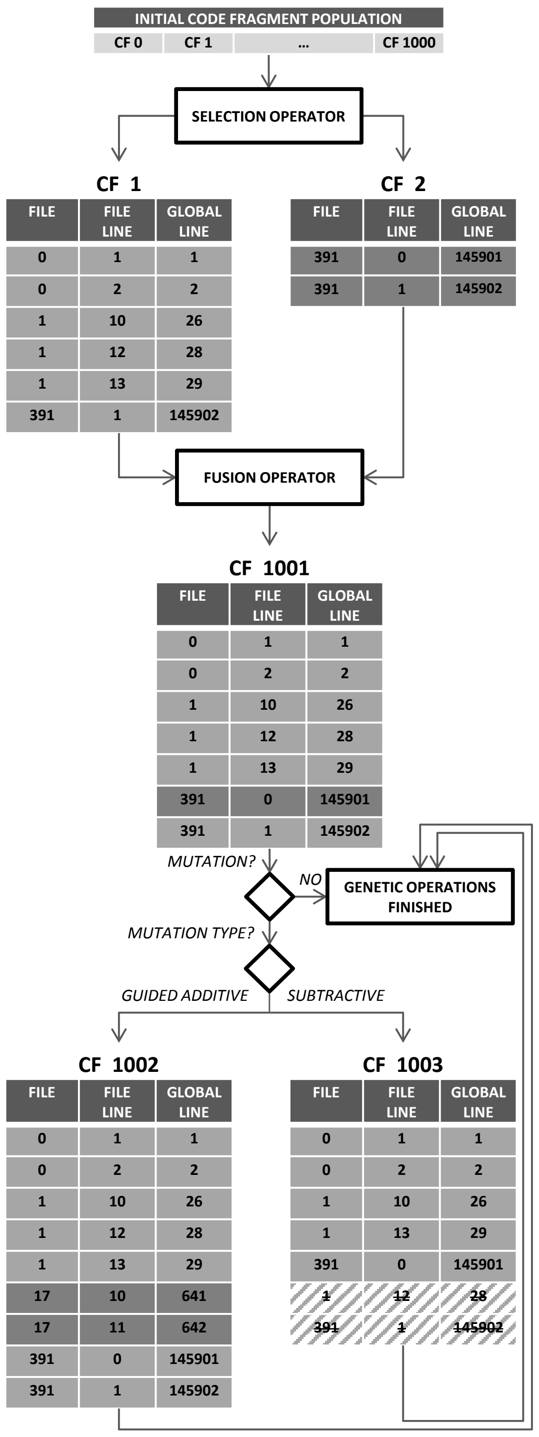
Prior to the application of the genetic operators, the best possible parents need to be selected from a given code fragment population so that the genetic operators have data to work with. For that reason, our approach uses a selection operator that is based on the widely used wheel selection method [9]: every code fragment in the population has a probability of being selected to reproduce that is proportional to its fitness value; therefore a higher fitness value implies a higher probability of being chosen to generate new code fragments.

5.1. Fusion

The fusion operator works like traditional evolutionary computation methods, in which a new individual is generated by combining the generic material of two existing individuals. Depending on the resulting combination and the environmental conditions, the new individual could outperform their parents or it may not even survive (or, subsequently, reproduce). In our approach, code fragments act as the reproducing individuals and the fusion operator is responsible for mixing the lines that they contain. As a result, the new code fragment contains a code line set in which the lines are ordered according to their relative position in the complete VGSC, without repetitions, in the case of lines that are present in both parents.

The fusion operator receives two code fragments and creates a new one by combining every code line in the parents, hence, without losing information. Most common recombination techniques in genetic algorithms such as crossover [10] imply inheriting parental content, partially or totally. The fusion operator could be considered as a special case of crossover that does not discard code lines from parent code fragments that have been selected as standing out and contain supposedly valuable information that is related to a requirement.

A typical case of fusion application in our CODFREL approach is shown in Fig. 2 with code fragment examples. The fusion operator takes two code fragments as parents (CF1 and CF2). First, the selection operator uses the wheel selection method to choose two parents. Then, the fusion operator generates a new code fragment containing the full code line sets that are present in its parents. Therefore, the new code fragment generated contains every code line in the parents, without repetitions (e.g., code line 145902 is not duplicated). The new individuals obtained with this operator may include a higher num-



CF 1

FILE	FILE LINE	GLOBAL LINE
0	1	1
0	2	2
1	10	26
1	12	28
1	13	29
391	1	145902

CF 2

FILE	FILE LINE	GLOBAL LINE
391	0	145901
391	1	145902

CF 1001

FILE	FILE LINE	GLOBAL LINE
0	1	1
0	2	2
1	10	26
1	12	28
1	13	29
391	0	145901
391	1	145902

CF 1002

FILE	FILE LINE	GLOBAL LINE
0	1	1
0	2	2
1	10	26
1	12	28
1	13	29
17	10	641
17	11	642
391	0	145901
391	1	145902

CF 1003

FILE	FILE LINE	GLOBAL LINE
0	1	1
0	2	2
1	10	26
1	13	29
391	0	145901
~~1~~	~~12~~	~~28~~
~~391~~	~~1~~	~~145902~~

Figure 2: Genetic Operators: Guided Mutation and Fusion over Code Fragments

ber of code lines than their parents, as shown in Fig. 2.

5.2. Mutation

The mutation operator makes new individuals show changes in their genetic material that are caused by random factors and are not due to being inherited from their parents. The (usually) minimal variations modifying the inherited material may be positive or negative, depending on whether the mutation produces adaptive advantages or disabilities.

In our CODFREL approach, the mutation operator is applied on newly generated fragments after they have been produced by the fusion operator. However, mutations do not always occur and depend on a probability of a new code fragment being affected by a mutation. After a mutation takes place, the new code fragment may contain new lines belonging to the video game code or lose some of the lines that were featured prior to the mutation. Our approach proposes that, since the search space is huge, the creation of good starting code fragments should be guided, since it is difficult to create or improve individuals by adding random code lines from the VGSC due to its size. However, our CODFREL approach proposes that the loss of code lines is not guided, so that even the removal of lines with high similarity with a requirement could lead to code fragments that are closer to the realization of such a requirement. This proposal regarding mutations could be improved or modified in future works, taking into account the results obtained. In the end, the mutation operators provide another possible solution for the target requirement (the unmodified code fragment and its parents are also possible solutions). The nature of the mutation applied by the operator is based on random criteria:

- **Subtractive Mutation**: This type makes the mutation operator remove lines from the code fragment by randomly selecting them, as shown in the bottom right section in Fig. 2. The number of lines removed and the selection criteria are random, so there is no need to ensure that the lines removed are consecutive.

- **Guided Additive Mutation**: When the operator performs an additive mutation, a random number of lines belonging to the original VGSC (and not necessarily consecutive) is added to the code fragment, first selecting a starting code line that acts as a reference for the eventual code line set added. The guidance consists in having a certain probability of selecting a starting code line in the VGSC that is not completely random; instead, the operator may select a starting code line featuring terms that are tagged as keywords in the requirement (and, eventually, more code lines surrounding that line). Fig. 2 shows an example featuring a new code fragment, CF 1002, that was created by applying this mutation operation to CF 1001. The code lines added are 641 and 642, with 642 being the starting code line featuring a keyword:

  – `641:  if ( units[i]->IsIdle() ) {`

  – `642:  units[i]->Reset` $\boxed{\texttt{Armour}}$ `Interface();`

  Code line 641, however, is a line that was added for being adjacent to 642 in the VGSC, like other lines which were not selected (but could have been) in the example.

6. Fitness of the CODFREL Approach

In the context of our CODFREL approach, the fitness step is intended to determine the value of each code fragment generated. Following the principles applied in evolutionary algorithms, the fitness step takes a code fragment population as input and measures the degree of adaptation of each code fragment to the environment (the adaptation being the quality of the code fragment as a solution for the requirement described). Once the step finishes, it provides a ranking in which every code is placed according to the fitness value assigned so that the top-ranked code fragments are the most similar to the requirement.

This similarity is evaluated in our approach through Latent Semantic Indexing (LSI) [7], which is currently the best performing Information Retrieval (IR) technique in terms of outcomes [11, 12, 13, 14]. In this step, it is responsible for comparing the code fragments proposed for a requirement with the requirement specified in natural language. In the context of LSI, the input for which the solution is searched is denoted as "query", while the individual elements to be evaluated as possible solutions for realizing this query are denoted as "documents". For our approach, this scheme implies that the requirement acts as a query, while code fragments are documents.

6.1. Natural Language Processing

Before applying LSI, the requirement is processed through well-known Natural Language Processing techniques (Part-of-Speech tagging [15] and Lemmatizing techniques [16]) so that the gap between the code fragment texts and the requirement is reduced.

- The requirement is first divided into words or **tokens**. Depending on the text complexity, separators such as spaces or semicolons may work as tokenizers (i.e., elements used to split strings), but descriptions involving in-code elements might require more sophisticated processing.

- The second stage in the requirement processing consists in removing articles, conjunctions, and other elements that do not provide useful information. This task is carried out by applying the **POS** (Part of Speech) technique. This tagging technique analyzes the grammatical role of the words in the text and helps remove the undesired material.

- Through the usage of semantic techniques such as Lemmatizing, words can be reduced to their semantic roots or **lemmas**. Thanks to lemmas, the language used in the requirement is unified, thus avoiding verb tenses, noun plurals, and strange word forms that negatively interfere with the fitness.

Fig. 3 shows the application of these techniques to a requirement. The token extraction step involves the use of separators. After the POS step, tokens like "a", "and", or "it" are removed for not being

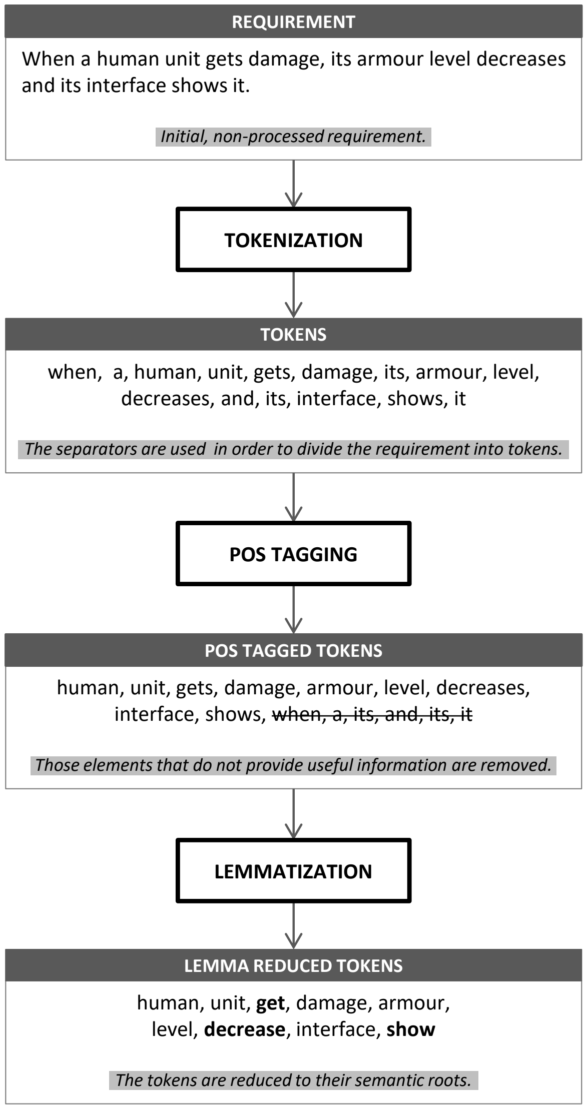

REQUIREMENT
When a human unit gets damage, its armour level decreases and its interface shows it.
Initial, non-processed requirement.
TOKENIZATION
TOKENS
when, a, human, unit, gets, damage, its, armour, level, decreases, and, its, interface, shows, it
The separators are used in order to divide the requirement into tokens.
POS TAGGING
POS TAGGED TOKENS
human, unit, gets, damage, armour, level, decreases, interface, shows, ~~when, a, its, and, its, it~~
Those elements that do not provide useful information are removed.
LEMMATIZATION
LEMMA REDUCED TOKENS
human, unit, get, damage, armour, level, decrease, interface, show
The tokens are reduced to their semantic roots.

Figure 3: Natural Language Processing Techniques

relevant in terms of substantial information. Lemmatizing analyzes and reduces words, transforming verb tenses such as "decreases" into "decrease".

The same Natural Language Processing techniques used with requirements are applied to code fragments, but include an additional step. This step involves removing stop words, which are programming language reserved words. Once a code fragment is processed, it contains terms such as variable and method names or words that are present in comments. The following example shows a set of two code lines as well as the result once it is processed:

- Code Lines:
  - `3107: int Unit::GetNumberOfWeapons() {`
  - `3108: return moduleWeapons->GetSize();`
- Result:
  - `unit, get, number, weapon, module, weapon, get, size`

The result obtained illustrates the techniques described above. Tokenizing makes use of separators (e.g., colons or spaces) and other criteria such as naming conventions (CamelCase, in the example) to extract tokens. POS removes elements that are not useful (e.g., "of", a conjunction that does not provide relevant content). Lemma extraction is shown by "weapon", which is a reduced, singular form. Finally, the additional step mentioned above searches for stop words to be filtered. Therefore, terms such as "int" or "return" will be removed since they are both defined as reserved words by the programming language used.

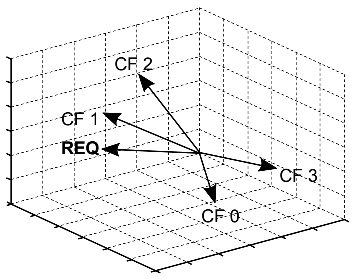
It is assumed that the language used in both the requirement and the code fragment should be similar in order to make the LSI work properly. If those languages are significantly different, not even the Natural Language Processing techniques will prevent the fitness from failing unless the user assists the process manually.

6.2. Latent Semantic Indexing (LSI) Fitness

After the Natural Language Processing, a co-occurrence matrix is built in order to represent terms in rows and each code fragment in a column, thus providing occurrence counters for every term in each code fragment or in the requirement. In the end, our LSI fitness uses a term set that defines the rows in the co-occurrence matrix, which is a union of two

	CODE FRAGMENTS					
TERMS	CF 0	CF 1	CF 2	CF 3	...	REQUIREMENT
HUMAN	0	1	0	0	...	1
UNIT	0	2	0	2	...	1
DAMAGE	0	1	0	0	...	1
DECREASE	0	2	1	0	...	1
ARMOUR	0	2	1	0	...	1
INTERFACE	0	1	1	0	...	1
SHOW	0	0	0	0	...	1
...	...	...	...	...	...	...

LSI RESULTS



RANKING	
CF 1	0.75
CF 2	0.50
CF 0	0.35
CF 3	0.15
...	

Figure 4: Fitness Operation via Latent Semantic Indexing (LSI)

sub-sets: the terms in the requirement and the terms in the code fragments.

The top part of Fig. 4 shows a schematic view of a co-occurrence matrix in our running example. The rows represent the terms from both the code fragments and the requirement. The columns represent the code fragments and the requirement. The values in the cells are the number of occurrences for each term in the code fragments and the requirements.

The co-occurrence matrix must be analyzed in order to elaborate the code-fragment ranking. First, it is normalized and decomposed through Singular Value Decomposition (SVD) [7], which factorizes the matrix and provides a vector set that represents the latent semantic value for every code fragment and the requirement. Similarity is evaluated using the angles formed by such multidimensional vectors, since cosine is a measure of similarity that is 1.0 for identical vectors and 0.0 for orthogonal vectors[17]. In the end, the cosine between each code fragment vector and the requirement vector is a value in the interval [-1, 1] that defines the similarity or proximity to the requirement, which allows a code-fragment ranking to be established.

Let C_1 be a code fragment in the population; let X be the vector representing the latent semantic value of C_1; let Y be the vector representing the latent semantic value of the requirement; the angle between X and Y is θ. The following expression defines the fitness function:

$$fitness(C_1) = \cos(\theta) = \frac{X \cdot Y}{\|X\| \cdot \|Y\|} \tag{1}$$

The bottom part of Fig. 4 shows a three-dimensional graph of the LSI results. The graph shows the representation of each one of the vectors, which are labeled with letters that represent the names of the code fragments. Finally, after the cosines are calculated, a value for each of the code fragments is obtained, indicating its similarity with the requirement.

7. Evaluation

This section presents the evaluation of our approach: the research questions, the oracle preparation, the experimental setup, the implementation details and the results obtained.

7.1. Research Questions

The following research questions address the evaluation of our approach considering different configurations and how they affect the results obtained. These questions make reference to a threshold, which is the number of complete methods (or code lines, depending on the granularity used) selected as a possible solution by Regular-LSI, the baseline approach. In addition, the research questions take into account the use of code fragments or method granularity by our approach:

RQ$_1$: *Does our CODFREL approach perform better than Regular-LSI when Regular-LSI uses a threshold value (the number of complete methods) of 1?*

Table 1: Configurations used in our CODFREL approach and Regular-LSI for the different research questions

	CODFREL	**Regular-LSI**
Research Question 1	Best Code Fragment	Best Complete Method
Research Question 2	Best Code Fragment	10 Best Complete Methods
Research Question 3	Best Code Fragment with Method Granularity	Best Complete Method
Research Question 4	Best Code Fragment with Method Granularity	10 Best Complete Methods
Research Question 5	Best Code Fragment	10 Best Code Lines

RQ$_2$: *Does our CODFREL approach perform better than Regular-LSI if the threshold value is the most widely used?*

RQ$_3$: *Does our CODFREL approach perform better than Regular-LSI if the threshold value is 1 and CODFREL uses method granularity?*

RQ$_4$: *Does our CODFREL approach perform better than Regular-LSI if the threshold value is the most widely used and CODFREL uses method granularity?*

RQ$_5$: *Does our CODFREL approach perform better than Regular-LSI when both use code line granularity (Regular-LSI considers each code line as a document) and the threshold value for Regular-LSI is the most widely used?*

Table 1 shows how the different configurations for our CODFREL approach and Regular-LSI combine according to each research question.

7.2. Oracle

The concept of oracle, in the context of our work, is applied to code line sets corresponding to the ground truth or full coverage for a requirement. In other words, this code line set represents the most accurate possible solution corresponding to a requirement. Twenty requirements, as well as the corresponding oracles, are provided by the game development company responsible for the design of Kromaia. Figs. 5 and 7 show key data regarding the requirement set, although detailed source code information is confidential. The requirements in the set were selected and provided by the developers for being a representative collection in terms of maintenance, and such selection criteria did not depend on dispersion within the VGSC. We perform a fair comparison by configuring the different versions (code line and method granularity) of our approach and Regular-LSI with the same requirement set mentioned, to ensure that neither Regular-LSI nor CODFREL are studied for optimized data sets.

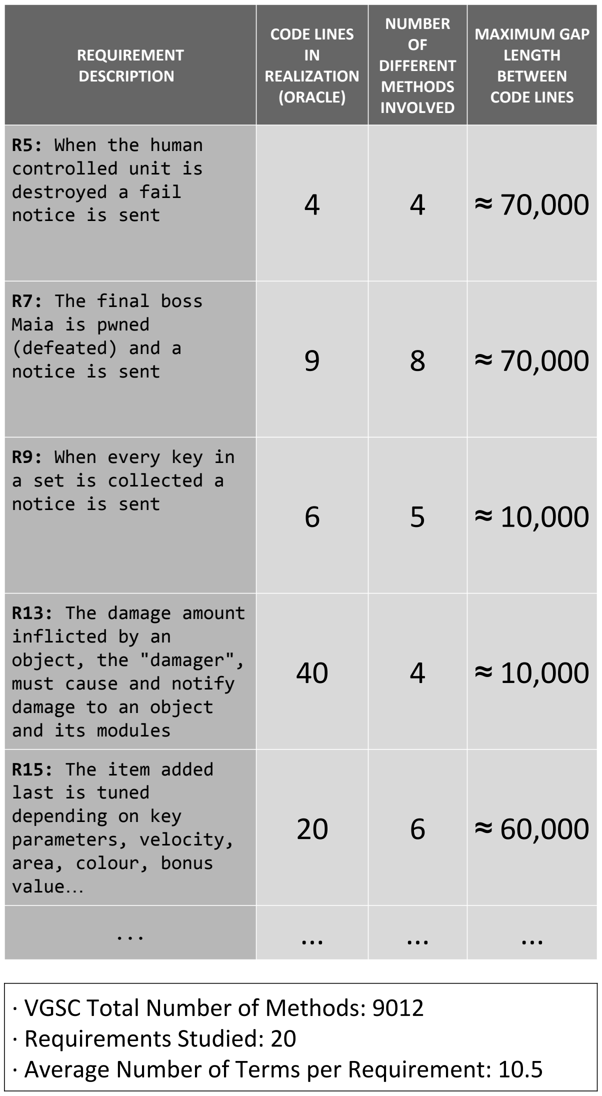
7.3. Experimental Setup

The evaluation measures the performance achieved by our approach. In addition, we compare our approach with a baseline approach (Regular-LSI) that achieves the best results in the literature [6]. Regular-LSI selects the method that is most relevant to the requirement by means of LSI. The LSI documents are methods, and the query is the requirement.

The first step involves feeding both our CODFREL approach and Regular-LSI with each of the requirements. Since CODFREL is not a deterministic approach, our approach features 30 runs for each requirement as suggested in [18], whereas Regular-LSI, which is deterministic, features one run.

Once our approach and Regular-LSI finish processing a requirement, they provide the solutions found for this requirement. Our CODFREL approach supplies a code fragment from various code lines that are present in the VGSC. Regular-LSI provides a solution consisting of a set of code lines for a complete method.

The results obtained by our approach and Regular-LSI are compared to the oracle through a confusion matrix, or error matrix [19]. Confusion matrices are used to study the performance achieved by a classification system on a certain test data set. The oracle



REQUIREMENT DESCRIPTION	CODE LINES IN REALIZATION (ORACLE)	NUMBER OF DIFFERENT METHODS INVOLVED	MAXIMUM GAP LENGTH BETWEEN CODE LINES
R5: When the human controlled unit is destroyed a fail notice is sent	4	4	≈ 70,000
R7: The final boss Maia is pwned (defeated) and a notice is sent	9	8	≈ 70,000
R9: When every key in a set is collected a notice is sent	6	5	≈ 10,000
R13: The damage amount inflicted by an object, the "damager", must cause and notify damage to an object and its modules	40	4	≈ 10,000
R15: The item added last is tuned depending on key parameters, velocity, area, colour, bonus value...	20	6	≈ 60,000
...	...	...	...

· VGSC Total Number of Methods: 9012
· Requirements Studied: 20
· Average Number of Terms per Requirement: 10.5

Figure 5: Sample containing some of the requirements in the case study and data relative to the methods involved.

indicates which data in that set is true or false. Confusion matrices are useful for evaluating the results provided by an approach and the ground truth that the oracle represents.

The confusion matrix arranges the results of the comparison into different categories:

- True Positives (TP), which refer to the number of code lines in the code fragment selected as a solution that are also present in the oracle.
- False Positives (FP), which denote the number of code lines that are present in the proposed solution but are not present in the oracle.
- False Negatives (FN), which denote the number of code lines present in the oracle that are not present in the code fragment or complete method marked as a solution.

Then, performance measurements are derived from the values in the confusion matrix. Specifically, we create a report that includes three performance measurements (precision, recall, and the F-measure).

- **Precision** is the fraction of correct code lines among the code lines selected, according to the corresponding oracle in the result proposed as a solution.

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

- **Recall** measures the number of code lines in the oracle that are correctly retrieved over the total number of code lines proposed in that solution.

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

- The **F-measure** corresponds to the harmonic mean of precision and recall. It is used to evaluate accuracy and is defined as follows:

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

7.4. Implementation Details

We have used the following libraries to implement the approach of this work: the OpenNLP Toolkit for the Processing of Natural Language Text [20] to develop the Natural Language Processing operations; and the Efficient Java Matrix Library (EJML) [21] to perform LSI and SVD. The computer used in the evaluation was a Toshiba Satellite PRo L830 laptop, with an Intel(R) Core(TM) i5-3317U@1.7GHz processor with 4GB RAM and Windows 8 64-bit.

The fusion operation is applied with a fusion probability (p_f). Through tuning tests, the value of p_f varied changing from preliminary values, like 0.5, to 1 in order to maximize the number of new individuals produced by fusion in each iteration of the algorithm (e.g., the change mentioned would involve not only duplicating the individuals created through fusion, but also duplicating the number of candidates to produce additional, mutated individuals). Additional research could improve parameters like this. The mutation operation is applied with a probability (p_m) of 0.25. The rest of the settings are detailed in Table 2. The focus of this paper is not to tune the values to improve the performance of our approach when applied to a specific problem. As suggested by Arcuri and Fraser [18], default values are good enough to measure the performance of search-based techniques in the context of testing. Nevertheless, we plan to evaluate all of the parameters of our approach in a future work. First, we started with default values used in the literature regarding software model feature traceability [22]. However, since the objective in this work is different (fine-grained requirement traceability in source code) and we use genetic operations to manipulate code fragments, the parameters are the result of starting with those studied in the literature and then performing preliminary tuning experiments. With the current configuration, 7 (μ) parents are combined in pairs to create 21 new code fragments. Apart from those new code fragments and depending on p_m, up to 21 mutated code fragments could also be created. Therefore, it is necessary to discard code fragments from the population after each iteration in order to keep the population stable, with 42 (r) being the maximum number of candidates (those with the lower fitness values) that could be removed.

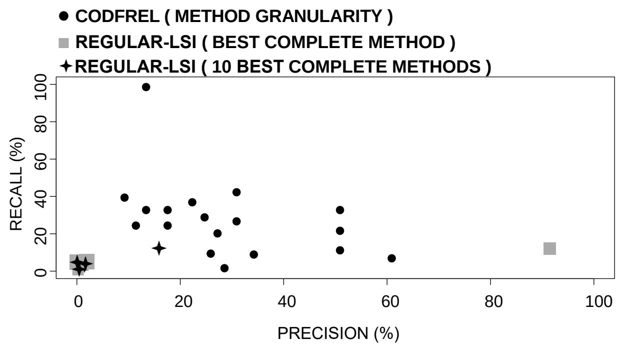
Table 2: CODFREL configuration parameters

Parameter description	**Value**
Size: Population size	1000
μ: Number of parents	7
λ: Number of offspring from μ parents	21
r: Maximum number of solutions replaced to stabilize population	42
p_f: Fusion probability	1
p_m: Mutation probability	0.25

In general, there are two atomic performance measures for search algorithms: one regarding solution quality and one regarding algorithm speed or search effort. In this paper, we focus on the solution quality. Therefore, we allocated a fixed amount of wall clock time for each of the runs of our approach. First, we ran some prior tests to determine the time needed to converge, and then we selected the budget time based on those tests. The allocated budget time was 1200 seconds. A prototype of CODFREL can be found at bitbucket.org/svitusj/SCoFBReL

7.5. Results

In this subsection, we present the results obtained for the case study in our approach and for Regular-LSI. Fig. 6 shows the charts with the recall and precision results. For CODFREL, a dot in the graph represents the average result (after 30 runs and due to random factors) of precision and recall for each of the 20 requirements. In the case of Regular-LSI, the solution for a requirement is deterministic, so there are no repetitions and the dots in the graph represent the results after a single run for each requirement. Table 3 shows the precision, recall, and F-measure mean values obtained for the case study by our approach and Regular-LSI.



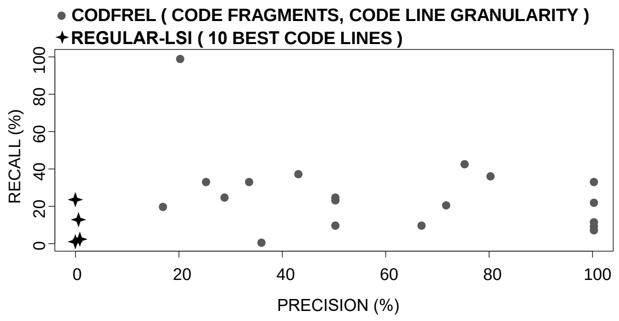


Figure 6: Results for our CODFREL approach and Regular-LSI with different configurations

Table 3: Precision, Recall, and F-Measure mean values and standard deviations for the case study

	Precision± (σ) (%)	**Recall**± (σ) (%)	**F-measure**± (σ) (%)
CODFREL	57±30	27±20	29±13
CODFREL, Method Granularity	28±15	27±20	21±9
Regular-LSI, Best Complete Method	4±19	0.5±2	0.9±4
Regular-LSI, 10 Best Complete Methods	0.7±3	0.5±2	0.6±2
Regular-LSI, 10 Best Code Lines	0.1±0.2	9±11	0.2±0.4

7.6. Research Question 1

To answer the first research question, it is necessary to study our CODFREL approach with code fragments and a Regular-LSI configuration with a threshold value of 1, since it only selects the best complete method.

RQ$_1$ answer. Fig. 6 and Table 3 show that CODFREL outperforms Regular-LSI in precision, recall, and F-measure, with average values of 57% in precision, 27% in recall, and 29% in F-measure. Regular-LSI obtained average values of 4%, 0.5%, and 9% in precision, recall, and F-measure, respectively, for the requirement set studied. The deviations for Regular-LSI, an approach which is deterministic, occur due to performance variations that are related to the different requirements and are not caused by different runs.

7.7. Research Question 2

The second research question takes into account that, while human subjects usually do not focus on a single candidate, they tend to consider no more than 10 candidate trace links [6]. In this case, our CODFREL approach uses code fragments and Regular-LSI selects the 10 best complete methods.

RQ$_2$ answer. Fig. 6 and Table 3 show that CODFREL obtains better results in recall, precision, and F-measure, with values of 27%, 57%, and 29%, respectively. In comparison, Regular-LSI gets significantly lower results for recall (0.5%), precision (0.7%), and F-measure (0.6%).

7.8. Research Question 3

The third research question studies a Regular-LSI configuration that only selects the best method and our CODFREL approach with method granularity.

This variation implies that CODFREL creates and manipulates code fragments that include every complete method from which code lines were selected, instead of using code line granularity.

RQ$_3$ answer. Fig. 6 and Table 3 show that CODFREL, with method granularity, outperforms Regular-LSI. Due to the granularity configuration used, precision does not reach 30%, with an average value of 28%, and recall and F-measure values are 27% and 21%, respectively.

7.9. Research Question 4

The fourth research question compares the following configurations: our approach CODFREL, with method granularity and Regular-LSI, with a threshold of 10 complete methods.

RQ$_4$ answer. Fig. 6 and Table 3 show that CODFREL, with method granularity, gets better results than Regular-LSI. Method granularity does have a remarkable impact on precision for our approach, with a value of 28%. However, since the values obtained by Regular-LSI are very low, the difference in the results is high, with this configuration of Regular-LSI selecting the 10 best complete methods as possible solutions.

7.10. Research Question 5

The fifth research question considers these configurations: our approach CODFREL with code fragments, and Regular-LSI with code line granularity which involves treating each line as a method using a threshold of 10 code lines.

RQ$_5$ answer. Fig. 6 and Table 3 show that CODFREL, with code fragments, gets better results than Regular-LSI configured with code line granularity. The granularity used by Regular-LSI significantly affects precision, with a value of 0.1%. Regular-LSI obtains values that are low in recall and in F-measure: 9% and 0.2%, respectively.

8. Discussion

The requirements were provided by one of the developers of Kromaia before we started this research work. The requirement selection criteria used by the developer did not depend on the requirement implementation being condensed in just one complete method or featuring a high dispersion level. In fact, the criteria involved the selection of requirements that were relevant in terms of maintenance. Most of the requirements provided were dispersed, with the average number of methods for a dispersed requirement being around 5.3. Fig. 5 shows that, even for requirements that include less than five lines, according to the oracles, it is not uncommon to find four or more methods involved. In addition, Fig. 7 shows that, in terms of cohesion, the realizations of the requirements studied include gaps between code lines of up to 70,000 code lines. This suggests that dispersed requirements should not be neglected in maintenance tasks.

Regular-LSI, which was selected for currently being the best performing IR technique, does not work properly for highly dispersed requirements in the VGSC of the case study, and the average results obtained are not good in the performance measures studied. The main cause for this is the fact that Regular-LSI works in terms of complete methods to trace requirements that are dispersed in various methods. Besides that, in those cases featuring a method that combines code lines that are relevant to the requisite with code lines that are not relevant, the irrelevant code lines prevent the method from getting a higher score. For instance, in the following requirement:

- `R1: When a human unit gets damage, its armour level decreases and its interface shows it.`

There is an event (gets damage) that has a noticeable impact on an entity known as a human unit in different contexts: the internal logics and structure in the entity, which need to be modified; and the interface elements that are directly related to this entity, which provide visual feedback regarding its internal status. This requisite is dispersed in five methods, each one of which only features an average of 13% (maximum 33%; minimum 8%) of relevant code lines for R1.

Our CODFREL approach outperforms Regular-LSI thanks to the fact that it uses code fragments. The total number of possible code fragments ($2^{145,000}$) makes a thorough exploration and evaluation unfeasible. The use of an evolutionary algorithm, however, allows our approach to explore the search space, and it gets better results than Regular-LSI that is configured to work with code line granularity, due to the size of the search space and high requirement dispersion. The results show that it is possible to use the proposed encoding and genetic operations in commercial software products that are similar to Kromaia. In each iteration of the evolutionary algorithm, the new code fragments created with the fusion operator are progressively larger, but code fragments that are remarkably large are prone to being discarded if the code lines accumulated do not contribute to increasing the value given to those code fragments.

There is another factor to be taken into account regarding the results: the use of keywords. There are terms featured in requirements, which, in spite of being relevant for such requirements, are ambiguous. In the requirement R1, terms like "decrease", "armour", and "level" are relevant for the requirement. "Level", however, is widely used and accepted as a video game Domain-Specific Language term with different but equally valid meanings: "level" could be considered as a stage or zone that should be cleared by the player, but it could also refer to the current player status. For that reason, "level" is relevant but ambiguous.

However, terms that are relevant but ambiguous are not discarded in our approach since they are used (along with the keywords) to calculate fitness values. In contrast, since keywords are terms that are both relevant and unambiguous, they are given more importance in our approach. They are not only used in fitness calculations but are also used to provide guidance in additive mutation. In R1, "decrease" and "armour" are suitable keywords that comprise the main concepts involved by the requirement. Additionally, we have considered the effects of guidance in subtractive mutation, which is an operator that, in our approach, removes code lines by randomly selecting them. We included a modified operator that takes keywords into account, like additive mutation, and tends to preserve code lines that include keywords. One possible disadvantage of this alternative operator is the low probability of removing code lines that contain keywords, even if such lines are not relevant; therefore, solution candidates with such content could not be improved through random modifications. In comparison to the results obtained with the first version of the operator, precision and recall increased an average of 20% in four of the 20 requirements studied. However, for 15% of the requirements the average results were 16% lower after using the new operator. This data could be studied in future works to produce a better subtractive mutation operator.

Even if CODFREL outperforms Regular-LSI, it does not achieve solutions that include every code line that is relevant for the requirements. Our analysis of the results suggests that tacit knowledge has a negative effect on the results.

Tacit knowledge is often assumed to be known by every domain expert. This assumption leads to a lack of documented presence of that knowledge, and requirements are no exception. The tacit knowledge related to the domain involved by requirements is usually considered and shared by the developers, who are responsible for the VGSC as well as for providing the requirements. In the end, every aspect of the domain knowledge (including tacit knowledge) that remains unwritten and the information provided by requirements are present in the VGSC. Therefore, requirements that do not feature a detailed description that reflects all relevant knowledge are incomplete, and the approaches searching for solutions will experience difficulties trying to find optimal results. The following example is a requirement that omits tacit knowledge that is actually present in the VGSC and the solution to be found:

- `R2: Shot input makes the human unit fire projectiles.`

This requirement omits relevant information regarding the modular nature of the VGSC featured by many entities. Units contain weaponry modules, which contain weapons. Besides, weapons internally manage a variable number of cannons, but only the cannons marked as "valid" are those responsible for ultimately firing projectiles.

Since tacit knowledge is the main issue to be studied in order to achieve better results with our approach, we plan to research it in more depth in our future works. In order to address this subject, we intend to use reformulation techniques so as to expand the requirements with elaborated descriptions provided by domain experts.

9. Future Work

The main issues to be addressed in future works are the eventual upgrade of genetic operators, the re-evaluation of parameters, and research on tacit knowledge:

With regard to genetic operators, it is possible that mutations involving the removal of code lines from code fragments should be guided, like additive mutations. In order to include such an operation, the management of terms that are not relevant, as opposed to keywords, could be useful.

The use of different values for the collection of parameters used in our approach could be studied in future works since tuning those values, like the fusion probability, the mutation probability, or the number of code fragments selected to be the parents of the next generation, could lead to configurations having an impact on the results and the time used. Due to the time required to test different configurations, future works should focus on this issue in particular.

Tacit knowledge is a key issue that should also be considered in future works. It would be necessary to expand the requirements available, and domain experts would be required for that task. The direct participation of domain experts would play a key role in providing additional explicit knowledge to be used in the guidance of the evolutionary algorithm in our CODFREL approach.

10. Threats to Validity

The classification for possible threats to validity in [23] reflects the necessary awareness regarding the limitations of our approach. This classification covers aspects that are related to both the approach itself and the commercial video game case study:

- **Internal Validity** refers to eventual issues related to existing causal relations. There is a possible risk for code line selection to be biased during the code fragment creation process due to the use of keywords in the population initialization as well as in guided additive mutations. That risk is reduced since our approach includes random deviation measures in these selection processes.

- **External Validity** is mainly concerned with the actual extent to which the results found can be generalized to case studies that are different from Kromaia, which is the commercial video game case study presented in this work. Two factors that increase the possibilities of generalization in the VGSC are the extensive use of a strict coding style and design patterns. These are widely used in real-time applications that are similar to video games. Nevertheless, our results should be replicated with other case studies before assuring their generalization.

- **Construct Validity** is an aspect worth taking into consideration since there is a risk that the operations involved in our approach may not accurately represent the desired functionalities for this research. The use of widely accepted measures such as precision and recall minimizes the risk described.

- **Reliability** deals with the possibility of the researchers influencing both the analysis process and the data used in the approach. In order to minimize the knowledge regarding the commercial video game case study, the VGSC comprehends a vast number of code lines, thus preventing requirements from being located too easily.

11. Related Work

A recent traceability survey [6] has identified the need for more industrial case studies. This work also shows that in spite of being the most commonly used, algebraic models (LSI and Vector Space Model) search in solution spaces with sizes below 2^{500}. Our work deals with an industrial case study that features a significantly wide space ($2^{145,000}$) to be explored and dispersed requirements, as shown in Fig. 7. Therefore, we work with code fragments instead of complete methods, and we search for possible solutions by means of an evolutionary algorithm since this type of algorithm has proven to be useful in large search spaces [24].

There are works that use design documents or domain models to support traceability [25] [26]. However, in the case study in this work, those artifacts were not available. Currently, video game developers are pressured by what is called "the age of crunch" [27] and the ever-increasing high demand of game content, which is caused by early access releases, post-launch updated versions, DLC (Downloadable Content), and games as a service. In this context, these artifacts end up not being synchronized or they are not even created, so approaches like CODFREL, which work when the artifacts mentioned are not available, are necessary.

The ADAMS document management system by De Lucia et al. [28] was used as evaluation context in trace recovery empirical experiments through LSI. Through case studies with students, as well as different controlled experiments, they have reinforced the empirical basis (De Lucia et al. [29], [30], [31]). Also, various studies by Cleland-Huang and colleagues, which are focused on Information Retrieval (IR)-based trace recovery, show the use of PIN-based retrieval as a model that supports the introduction of probabilistic trace recovery. This model was implemented in their tool, Poirot (Lin et al. [32]). To a great extent, their work focuses on accuracy improvements for their tool. The enhancements include the localization of key phrases [33], synonymy management with a thesaurus, and a glossary that weights the most important terms in the project with higher values [33].

In comparison to the works mentioned above, our work makes use of LSI to guide an evolutionary algorithm. LSI does not give values directly to complete artifacts such as methods but rather guides the exploration within the solution space.

The use of probabilistic models has been used by several researchers to support trace recovery. For instance, the probabilistic topic model Latent Dirichlet Allocation is one of the various IR models combined by Dekhtyar et al. [34] using a voting scheme. Abadi et al. [35] proposed using Probabilistic Latent Semantic Indexing and two information theory-based concepts: Jensen-Shannon Divergence and Sufficient Dimensionality Reduction. Parvathy et al. [36] suggested the Correlated Topic Model, while Getters et al. [37] proposed the Relational Topic Model. The use of a non-centralized set of self-organized agents that work together to extract conclusions is a swarm-like approach to trace recovery implementation proposed by Sultanov and Huffman Hayes [38]. Similarity calculations in the Cartesian plane are also possible. Capobianco et al. [39] suggested using B-Splines as Natural Language artifact representations so that similarity would be given by the distance between these splines on the Cartesian plane.

Unlike those works, our work searches for solutions in terms of code fragments. These solutions could be any code line set in the source code. Solutions of this kind are necessary in commercial contexts like the AAA video game industry, where requirements are dispersed across several methods.

Genetic Algorithms have been used to configure and assemble IR processes automatically in order to support different software engineering tasks [40], thereby positively affecting the time and resources spent on maintenance. These approaches determine near optimal solutions for the different IR process stages without training, and they outperform previous approaches without remarkable differences in comparison to combinatorial and supervised approaches. Our work also uses an evolutionary algorithm, but it explores in a completely different way. Instead of using the evolutionary algorithm to search for the best IR technique combination, we use the evolutionary algorithm to search the vast code fragment solution space, which is the case for AAA commercial video game software ($2^{145,000}$, in our case study).

Other works have studied the extent to which IR techniques can provide decision support in the context of large, industrial software engineering tasks in terms of traceability and maintenance [41]. In these works, the researchers noticed issues regarding

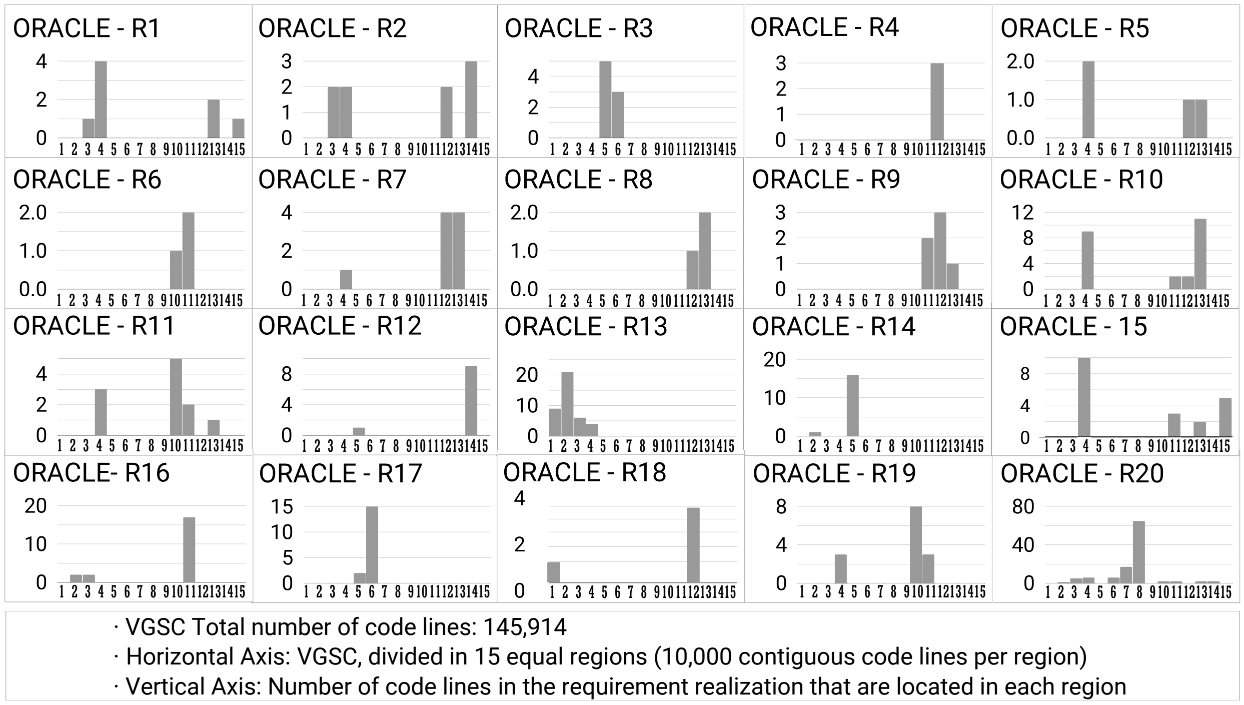


Figure 7: Dispersion of the requirement realizations in the source code of the case study.

the difficulties of scaling IR techniques to industry data, due to latent semantic analysis. The way in which IR-based traceability recovery tools are used by developers and how they validate/discard correct information and false positives has been studied in works that are focused on going beyond the performance analysis of IR-based traceability methods [42]. The approach used in those works suggests counting recovered traceability links in order to increase the quality of the validation process that is carried out by the users that are working with recovery tools. Our approach not only outperforms Regular-LSI, but it also finds a new cause behind the inability to obtain better requirement traceability results: tacit knowledge that is not formalized when the requirements are written.

Recent works [43, 44, 45] propose the use of Neural Networks to address the challenge of traceability. Guo et al. [43] leverage Word Embedding and Recurrent Neural Network (RNN) models to generate trace links, which contain the requirements artifact semantics and the domain knowledge. Zhao et al. [44] propose training deep neural networks to generate text-based knowledge in software repositories in order to improve the accuracy of TLR. The work in [45] presents some challenges in traceability and some of their proposals consider addressing these traceability issues through neural networks.

The above works based on Neural Networks require the existence of a knowledge base for training. For example, in [43], the training set is composed of 45% of the 769,366 artifacts, so this training set contains 423,151 feature vectors. However, some industrial companies do not store enough information to create the required knowledge base for Neural Networks. Actually, this lack of documented knowledge has been previously reported as the knowledge vaporization problem [46]. Nevertheless, these domains also need to recover the traceability links, and our CODFREL approach as well as the Regular-LSI approach can be applied for traceability recovery even without a knowledge base.

12. Conclusions

In comparison to those generated by Regular-LSI, the solutions provided by our approach are better starting points, assuming, however, that both approaches need to be refined manually. The use of evolutionary algorithms and code fragments improves the results obtained by Regular-LSI, since a VGSC like the one featured in the video game case study involves highly dispersed methods and a huge solution space. However, tacit knowledge, which is not explicitly present in the requirements, prevents our approach from achieving better solutions. Getting domain experts to be more involved could make this implicit knowledge become explicit. To facilitate the adoption of CODFREL, we have made a reference implementation freely available.

Acknowledgements

This work has been partially supported by the Ministry of Economy and Competitiveness (MINECO) through the Spanish National R+D+i Plan and ERDF funds under the Project ALPS (RTI2018-096411-B-I00).

References

[1] R. Watkins, M. Neal, Why and how of requirements tracing, IEEE Software 11 (1994) 104–106.

[2] A. Ghazarian, A research agenda for software reliability, IEEE Reliability Society 2009 Annual Technology Report (2010).

[3] O. C. Z. Gotel, C. W. Finkelstein, An analysis of the requirements traceability problem, in: Proc. IEEE Int. Conf. Requirements Engineering, 1994, pp. 94–101. doi:10.1109/ICRE.1994.292398.

[4] G. Spanoudakis, A. Zisman, Software traceability: a roadmap, in: Handbook of Software Engineering and Knowledge Engineering: Vol 3: Recent Advances, World Scientific, 2005, pp. 395–428.

[5] P. Rempel, P. Mäder, Preventing defects: The impact of requirements traceability completeness on software quality, IEEE Transactions on Software Engineering 43 (2017) 777–797.

[6] M. Borg, P. Runeson, A. Ardö, Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability, Empirical Software Engineering 19 (2014) 1565.

[7] T. K. Landauer, P. W. Foltz, D. Laham, An introduction to latent semantic analysis, Discourse processes 25 (1998) 259–284.

[8] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology Behind Search, 2nd ed., Addison-Wesley Publishing Company, USA, 2008.

[9] M. Affenzeller, S. Winkler, S. Wagner, A. Beham, Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications, 1st ed., Chapman & Hall/CRC, 2009.

[10] L. Davis, Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.

[11] M. Revelle, B. Dit, D. Poshyvanyk, Using data fusion and web mining to support feature location in software., in: ICPC, 2010, pp. 14–23.

[12] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, Feature location via information retrieval based filtering of a single scenario execution trace, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 234–243. URL: http://doi.acm.org/10.1145/1321631.1321667. doi:10.1145/1321631.1321667.

[13] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, V. Rajlich, Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval, IEEE Transactions on Software Engineering 33 (2007) 420–432.

[14] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process 25 (2013) 53–95.

[15] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, S. Panichella, On the role of the nouns in IR-based traceability recovery, in: Proc. IEEE 17th Int. Conf. Program Comprehension, 2009, pp. 148–157. doi:10.1109/ICPC.2009.5090038.

[16] J. Rubin, M. Chechik, A survey of feature location techniques, in: Domain Engineering, Springer, 2013, pp. 29–58.

[17] A. Singhal, et al., Modern information retrieval: A brief overview, IEEE Data Eng. Bull. 24 (2001) 35–43.

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

[19] S. V. Stehman, Selecting and interpreting measures of thematic classification accuracy, Remote Sensing of Environment 62 (1997) 77 – 89.

[20] Apache opennlp: Toolkit for the processing of natural language text, https://opennlp.apache.org/, 2017. [Online; accessed 12-November-2017].

[21] P. Abeles, Efficient java matrix library, http://ejml.org/, 2017. [Online; accessed 9-November-2017].

[22] J. Font, L. Arcega, Ø. Haugen, C. Cetina, Leveraging variability modeling to address metamodel revisions in model-based software product lines, Computer Languages, Systems & Structures 48 (2017) 20–38.

[23] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer, 2012.

[24] L. Arcega, J. Font, C. Cetina, Evolutionary algorithm for bug localization in the reconfigurations of models at runtime, 2018, pp. 90–100. doi:10.1145/3239372.3239392.

[25] J. Cleland-Huang, J. Huffman Hayes, J. Domel, Model-based traceability, 2009, pp. 6–10. doi:10.1109/TEFSE.2009.5069575.

[26] J. Guo, N. Monaikul, C. Plepel, J. Cleland-Huang, Towards an intelligent domain-specific traceability solution, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 755–766. URL: http://doi.acm.org/10.1145/2642937.2642970. doi:10.1145/2642937.2642970.

[27] IGDA, International Game Developers Association, 2018. URL: http://tiny.cc/ucev5y.

[28] A. De Lucia, F. Fasano, R. Oliveto, G. Tortora, Adams re-trace: A traceability recovery tool, in: Proc. Ninth European Conf. Software Maintenance and Reengineering, 2005, pp. 32–41. doi:10.1109/CSMR.2005.7.

[29] A. De Lucia, M. D. Penta, R. Oliveto, F. Zurolo, Coconut: Code comprehension nurturant using traceability, in: Proc. 22nd IEEE Int. Conf. Software Maintenance, 2006, pp. 274–275. doi:10.1109/ICSM.2006.19.

[30] A. De Lucia, F. Fasano, R. Oliveto, G. Tortora, Recovering traceability links in software artifact management systems using information retrieval methods, ACM Trans. Softw. Eng. Methodol. 16 (2007).

[31] A. De Lucia, R. Oliveto, G. Tortora, Assessing ir-based traceability recovery tools through controlled experiments, Empirical Software Engineering 14 (2009) 57–92.

[32] J. Lin, C. C. Lin, J. Cleland-Huang, R. Settimi, J. Amaya, G. Bedford, B. Berenbach, O. B. Khadra, C. Duan, X. Zou, Poirot: A distributed tool supporting enterprise-wide automated traceability, in: Proc. 14th IEEE Int. Requirements Engineering Conf. (RE'06), 2006, pp. 363–364. doi:10.1109/RE.2006.48.

[33] X. Zou, R. Settimi, J. Cleland-Huang, Improving automated requirements trace retrieval: a study of term-based enhancement methods, Empirical Software Engineering 15 (2010) 119–146.

[34] A. Dekhtyar, J. H. Hayes, S. Sundaram, A. Holbrook, O. Dekhtyar, Technique integration for requirements assessment, in: Proc. 15th IEEE Int. Requirements Engineering Conf. (RE 2007), 2007, pp. 141–150. doi:10.1109/RE.2007.17.

[35] A. Abadi, M. Nisenson, Y. Simionovici, A traceability technique for specifications, in: Proc. 16th IEEE Int. Conf. Program Comprehension, 2008, pp. 103–112. doi:10.1109/ICPC.2008.30.

[36] A. G. Parvathy, B. G. Vasudevan, R. Balakrishnan, A comparative study of document correlation techniques for traceability analysis, in: ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume ISAS-2, Barcelona, Spain, June 12-16, 2008, 2008, pp. 64–69.

[37] M. Gethers, R. Oliveto, D. Poshyvanyk, A. De Lucia, On integrating orthogonal information retrieval methods to improve traceability recovery, in: Proc. 27th IEEE Int. Conf. Software Maintenance (ICSM), 2011, pp. 133–142. doi:10.1109/ICSM.2011.6080780.

[38] H. Sultanov, J. H. Hayes, Application of swarm techniques to requirements engineering: Requirements tracing, in: Proc. 18th IEEE Int. Requirements Engineering Conf, 2010, pp. 211–220. doi:10.1109/RE.2010.33.

[39] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, S. Panichella, Traceability recovery using numerical analysis, in: Proc. 16th Working Conf. Reverse Engineering, 2009, pp. 195–204. doi:10.1109/WCRE.2009.14.

[40] B. Dit, Configuring and assembling information retrieval based solutions for software engineering tasks, in: Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2016, pp. 641–646. doi:10.1109/ICSME.2016.85.

[41] M. Unterkalmsteiner, T. Gorschek, R. Feldt, N. Lavesson, Large-scale information retrieval in software engineering - an experience report from industrial application, Empirical Software Engineering 21 (2016) 2324.

[42] G. Bavota, A. De Lucia, R. Oliveto, G. Tortora, Enhancing software artefact traceability recovery processes with link count information, Information and Software Technology 56 (2014) 163 – 182.

[43] J. Guo, J. Cheng, J. Cleland-Huang, Semantically enhanced software traceability using deep learning techniques, in: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 3–14. URL: https://doi.org/10.1109/ICSE.2017.9. doi:10.1109/ICSE.2017.9.

[44] Y. Zhao, T. S. Zaman, T. Yu, J. Huffman Hayes, Using deep learning to improve the accuracy of requirements to code traceability, in: Challenges of Traceability: The Next Ten Years, 2017, pp. 22–24.

[45] G. Antoniol, J. Cleland-Huang, , J. Huffman Hayes, M. Vierhauser, The grand challenges of traceability: The next ten years, 2017, pp. 23–49. doi:10.1109/ICSE.2017.9.

[46] J. S. van der Ven, A. G. J. Jansen, J. A. G. Nijhuis, J. Bosch, Design decisions : The bridge between rationale and architecture, 2006, pp. 4–5. doi:10.1007/978-3-540-30998-7_16.