Innovations in Systems and Software Engineering (2022) 18:193–213
https://doi.org/10.1007/s11334-021-00418-2

S.I. : SOFTWARE AND SYSTEMS REUSE

Requirements traceability recovery for the purpose of software reuse: an interactive genetic algorithm approach

Mohamed Salah Hamdi[1] · Adnane Ghannem[1] · Marouane Kessentini[2]

Received: 1 April 2021 / Accepted: 30 October 2021 / Published online: 1 December 2021
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2021

Abstract
Traceability allows engineers to trace and monitor the relationships between software artifacts. Monitoring these relationships is vital to many software engineering activities such as software understanding and reuse. Grasping these relationships is studied in the framework of Requirement Traceability Recovery (RTR). RTR is vital to software reuse as it allows the identification and comparison of requirements of new and existing systems, and hence the reuse of software system components. Due to the difficulties in recovering the traceability links manually, only few software development processes take the monitoring of these relationships fully into account. Many attempts to automate the RTR task that enjoyed some success are based on methods from the field of information retrieval. However, these methods only concentrate on calculating the textual similarity between various software artifacts and do not take into account other properties of the artifacts. In this paper, we propose a search-based RTR approach using genetic algorithms, that relies not only on semantic similarity between software artifacts, but also takes into account the history of reuse of the artifacts, and incorporates knowledge into RTR in the form of user (designer/developer) feedback. Experimental results show that the approach is promising.

Keywords Software reuse · Interactive genetic algorithm · Requirements engineering · Requirements traceability

1 Introduction

In software development, lots of information is generated. This information consists of artifacts such as source code, design documents, test cases, bug reports, and manual pages. Describing and following the life cycle of artifacts is called traceability. Traceability allows engineers to trace and monitor the relationships between artifacts. Monitoring these relationships is vital to many software engineering activities such as process compliance and product improvement, and software understanding and reuse. These relationships, or traceability links, can exist between design documents and requirements, source code and design documents, requirements and source code, requirements and test cases, manual pages and requirements, bug reports and manual pages, etc. In this work, we are focusing on the relationships between requirements and source code elements such as functions and classes. We could facilitate software reuse by taking into account the relationships between the requirements of a project and the code created during implementation. Grasping these relationships can help engineers in identifying the code elements that implement the requirements they want to reuse [1]. This is especially important because the developers are no longer as familiar with the code as during the initial development. Grasping these relationships is studied in the framework of Requirement Traceability Recovery (RTR), a topic that has attracted more and more attention in the field of software engineering.

In RTR, the focus is on re-/constructing traceability links between requirements and other software system artifacts. Here, we focus on links between requirements and source code elements. The importance of RTR has been well established in recent years [2–16]. RTR is vital to software reuse as it allows the identification and comparison of requirements

✉ Mohamed Salah Hamdi
mshamdi@abmmc.edu.qa

Adnane Ghannem
adnane.ghannem@abmmc.edu.qa

Marouane Kessentini
marouane@umich.edu

1 Information Systems Department, Ahmed Bin Mohammed Military College, Doha, Qatar

2 Computer and Information Science Department, University of Michigan, Ann Arbor, MI, USA

of new and existing systems, and hence the reuse of software system components.

When the industrial designer/developer is developing new software (system), he can obtain the benefits from the proposed work by using it to recover traceability links between the requirements of the new system and source code elements belonging to available old systems (e.g., systems developed earlier by the designer/developer). This will contribute to allowing systematic software reuse which will reduce the development cycle time and cost and improve the quality of software.

RTR is also of great importance to many other software engineering activities, such as verification, change impact analysis, program comprehension, risk analysis, impact analysis, criticality assessment, and test coverage analysis. RTR is also important in adaptive systems that change frequently and continually and need to manage the requirement changes and analyze their impact.

Unfortunately, performing RTR manually is person-power intensive, extremely tedious, error-prone, and time-consuming. Due to the difficulties in recovering the traceability links manually, only few software development processes take the monitoring of these relationships fully into account, and developers' dereliction is obvious. As a result, "RTR as a practice" is a not always adopted by developers [17, 18] and RTR is typically taken as afterthought.

Many studies attempted to use methods based on the field of Information Retrieval (IR) to automate the RTR task. The IR-based methods are classic ways for traceability recovery [7, 19, 20] that focus on calculating text similarity between software artifacts, creating traceability links between artifacts that have high similarity. Researchers have applied techniques such as Latent Semantic Indexing (LSI) [21], Vector Space Model (VSM) [22], and probabilistic models [23]. Several IR-based tools have been developed. Examples include REquirements TRacing On-target RETRO [24, 25], ReqSimile [26], Poirot:TraceMaker [27], ReqAnalyst [28], and ADAMS Re-Trace [29].

These IR-based methods have been shown to help in generating the traceability links automatically and to reduce the time for producing the traceability mapping and have enjoyed some success. However, these methods only concentrate on calculating the textual similarity between various software artifacts and do not take into account other properties of the artifacts. They typically only consider the text of software artifacts, which contains a lot of information that is not relevant for traceability. This results in lowering the accuracy of RTR [19, 21, 30].

Many studies attempted to address these issues and improve IR-based techniques for generating the traceability links. Some studies (e.g., [19, 31, 32]) proposed the use of structural information (relationships between source codes, like class inheritance) for improving the IR-based techniques. Such approaches faced the challenge of efficient and extensive investigation of the structural information of related artifacts [19]. Other studies (e.g., [18]) proposed to focus only on essential information for recovering traceability links based on the corresponding domain knowledge. Other attempts (e.g., [33]) suggested focusing on the importance of the vocabulary base used for tracing and on using secondary measures for assessing the traceability mappings.

In this paper, we propose a novel search-based RTR approach using genetic algorithms, that relies not only on semantic similarity between software artifacts, but also takes into account the history of reuse of the artifacts, and incorporates knowledge into RTR in the form of user (designer/developer) feedback.

In this work, we deal with natural language requirements, and the precision of the produced links depends on the textual similarity between requirements and source code elements. As source code naming can fall to abbreviation and/or agglomeration (e.g., via camel case conventions, common in Java) which would not be expected in natural language, requirements and source code elements can diverge from each other, which decreases their textual similarity.

We believe that through the exploitation of other the sources of information, namely the history of reuse of the artifacts, and the incorporation of user (designer/developer) feedback, we can build improved traceability recovery.

We aim at automating the RTR process and help designers/developers find reusable source code matching their requirements. Code reuse is a great goal, but often proves to be difficult due to many challenges such as the dependencies of the code element to be reused from its originating system that could inhibit its reuse. There are also many other traits of software quality that impact the reuse of a code element such as its security, reliability, performance efficiency, and maintainability.

Through RTR, we aim at contributing to the detection of code elements that are good candidates to match the requirements of the new system being developed and that surely need to undergo extension and adaptation to suit the new system. Of course, the quality of the software element plays a major role in how easily it can be reused. Code elements of good quality can be reused in new ways that differ substantially from the code's original design intent. It is clear that an investment of time and resources during development is needed for creating such good, truly reusable, code elements.

For this purpose, we consider the RTR process as a search problem. Although evolutionary search has been applied to many software engineering problems [34], it is not common to consider RTR as an optimization problem.

The approach takes as input a software system and a set of requirements for which appropriate reusable code elements need to be identified and generates a set of traceability links (i.e., a solution) between the requirements and source

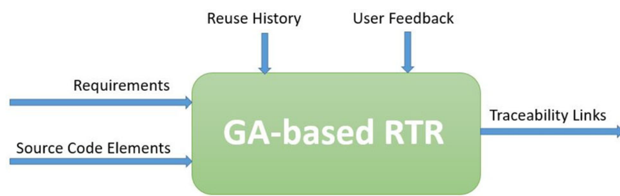


Fig. 1 Genetic algorithm-based requirements traceability recovery

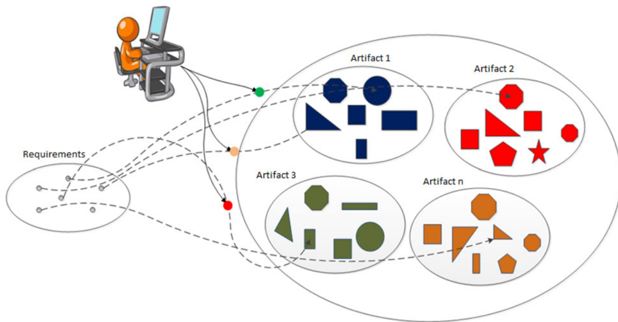


Fig. 2 Capturing feedback about detected traceability links between requirements and code element artifacts through interaction with the user

code elements (class, method, etc.) of the software system as shown in Fig. 1.

In the framework of a solution to the RTR problem, each requirement is assigned to one or many code elements of the software system. The assignment takes into account the *semantic similarity* between the documents, i.e., the texts, representing the requirement and the code element. The assignment also takes into account statistics from the *reuse history* of the software system. We use two types of measures, namely the *recency of reuse measure* and the *frequency of reuse measure.* The intuition behind the use of the frequency of reuse measure is that code elements that are reused more frequently than others are more likely to be reused now, i.e., are related to the new requirement at hand.

In this work, we assume the existence of a software reuse environment that plays an important role in exploiting reusable artifacts. Environments that support reusing software artifacts are of great interest [35]. For an effective reuse of code elements, we assume that the reuse environment stores information about the reuse history of individual code elements, namely information about the recency of reuse (last time of reuse of the element) and frequency of reuse (number of times the element was reused).

To make the assignment of requirements to code elements more informed, we also take into account the *user (designer/developer) feedback*. Figure 2 sketches the integration of the domain-specific knowledge (in form of user feedback) through interaction with the user. The user (designer/developer) can provide feedback whenever asked to do so.

We evaluate the approach using three object-oriented open-source projects. In a first set of experiments, we perform the evaluation by running a Genetic Algorithm (GA) to conduct RTR automatically based only on semantic similarity and reuse history. In a second set of experiments, we incorporate additionally user (designer/developer) knowledge in the form of feedback using an Interactive Genetic Algorithm (IGA). We conduct this evaluation using two types of feedback metrics, namely binary feedback and a more granulated feedback using ratings in the number range [0, 0.3, 0.5, 0.8, 1].

The rest of this paper is organized as follows. Section 2 presents the related work and provides a motivating example to illustrate the proposed work. Section 3 presents the RTR problem formally and analyzes its complexity. Section 4 discusses the modeling the RTR problem using genetic algorithms. In Sect. 5, we present and discuss the experimental results. Sections 6 presents conclusions and future directions of work.

2 Background and motivation

2.1 Software reuse

Software reuse, introduced in 1968, is defined by Krueger [36] as "the process of creating software systems from existing software rather than building software systems from scratch." Frakes and Kang [37] define software reuse as "the use of existing software or software knowledge to construct new software." The basic insight behind software reuse is that most software systems are different variants of already build systems within certain domains and are not new [37]. Software reuse can be ad hoc, systematic, based on composition of code, based on generation of code, involving only code reuse, or involving reuse of all kinds of artifacts. Software reuse can focus on technical issues like programming language support, retrieval of artifacts, and artifact repositories, as well as on nontechnical issues like involvement of humans, initiation of reuse processes, and modification of non-reuse processes [38], in addition to organizational, economic, administrative, political, and psychological impediments [39, 40].

Although in the 1980s some researchers such as Krueger [36] believed that software reuse failed to become a standard software engineering practice, today, libraries, frameworks, and their APIs are commonplace and they are wholly purposed to reuse [41, 42] and there is still renewed interest in software reuse by many researchers [37, 38, 43–49]. These researchers think that traditional software engineering methods are inadequate, and that software reuse can help building bigger and more complex systems that can be delivered on time and that perform better with regard to reliability and

cost, providing, hence, a better way of doing software engineering. Reusing the outcomes of previous projects could increase the efficiency of software development. Although it is not clear how general these benefits could be, some researchers (e.g., [50–52]) think that source code reuse could improve programmer productivity. In a study conducted by Selby [53], the findings show that module reuse could reduce the average "total development time per source line" from 1.089 man hours to 0.047 man hours. Although this performance looks to be rather idealized, because software reuse requires finding the module to reuse, interfacing it with the current project, and possibly editing it, if reusable code is found, then the time saved could be impressive.

We could facilitate software reuse by taking into account the relationship between the requirements of a project and the code created during implementation. We need to monitor these relationships in order to be able to match the requirements of a new project to those recorded for past ones. Although monitoring these relationships is vital to many software engineering activities such as process compliance and product improvement, and software understanding and reuse, it is, unfortunately, person-power intensive, time-consuming, error-prone, and lacks tool support. Therefore, only few software development processes take it fully into account. Automating monitoring the relationships between requirements and source code elements, i.e., the automated generation of traceability links, could smooth the way for better solutions. We call this Requirement Traceability Recovery (RTR).

2.2 Requirement traceability recovery

During the development of a software system, many artifacts such as requirement specifications, design, source code, software analysis, and test models are created. Artifacts can generalize, refine, rationalize, evolve from, overlap with, satisfy, conflict with, or depend on other artifacts. Artifacts can also be linked to the stakeholders that participated in their creation. Establishing these relationships between the artifacts of a software system is called software traceability.

Software traceability allows describing the system from different perspectives and levels of abstraction and contributes to improving its understandability. Software traceability is important for all phases of software development. Traceability links can assist in executing and integrating system changes, in coverage and verification analysis, in validation, in system testing, in system inspection, and in system acceptance. Software traceability is crucial for checking the completeness and correctness of the system, for the quality of its final version, for its maintenance, and for finding reusable components in it [54–56].

Software traceability has been studied from many different aspects. Some studies such as [57–62] focused on the diversity among software traceability and on defining it properly. Other studies such as [63–67] looked to different ways for supporting the establishment of traceability relations. In some other studies such as [68–72], the emphasis was on developing methods for representing and maintaining traceability links. In some other work like [73–77], the focus was on using traceability relations in the development life cycle.

Although, many studies such as [54, 65, 78–81] attempted to perceive traceability as a concept that encompasses the whole software development process and do not put special emphasis on the requirements, the origins of traceability lie in the field of Requirements Engineering (RE). Requirements engineering is usually defined as an activity aiming at discovering, documenting, and maintaining a set of requirements. Since the term "engineering" is used, systematic and repeatable techniques need to be used to ensure the completeness, consistency, and relevance of the system requirements [82]. A large part of the research dealing with traceability has been done by researchers of this domain [83]. In many standards and guidelines for requirements engineering, the traceability of requirements is a concern [58, 84].

The focus of many researchers on traceability in the context of the specification of system's requirements led to the extensive use of the term requirements traceability [58, 77]. Requirements traceability is concerned with establishing traceability relations between requirements and other software artifacts. Requirements traceability can be interesting for stakeholders in different ways. An end user, for example, may want to know the design objects that satisfy certain requirements. A designer, on the other hand, may want to know the constraints related to a given design object. The constraints are defined by the requirements.

In software projects, the traceability of requirements should be planned so that the people involved in the project can take advantage of it in an efficient manner, and so that the productivity of the project team is not influenced negatively. The appropriate degree or extent of traceability allows keeping an eye on implemented requirements and on the relevant artifacts (e.g., models, source code, test cases, etc.) that can be traced back to them. The right degree of traceability needed in a given software project depends on many factors such as the actual intended benefit of traceability, the cost, the project complexity, the possibility of assigning requirements to artifacts, and the tools used during the software development process.

In requirements traceability, relationships between requirements and artifacts are defined and maintained. Requirements traceability can be performed by writing and updating cross-reference and indexing schemes or traceability matrices manually. Many methods, environments, and tools assume the manual identification of traceability relations [70, 71, 85, 86]. The manual recovery of requirements traceability links in large systems can be costly and

may overlook links. This can make organizations reluctant to enforce requirements traceability. Therefore, a growing body of research is investigating automatic Requirements Traceability Recovery (RTR), i.e., the automatic recovery of traceability links between requirements and other artifacts (e.g., [87–91]). The automatic recovery of links can rely on information about artifacts and other links given a set of inference rules [62, 92, 93]. As text is the common form of information representation, many automated approaches rely on information retrieval techniques. These approaches use the similarity between pairs of artifacts to recover links between them [94, 95]. These approaches cannot identify traceability links with a rich semantic meaning when those pairs do not contain overlapping or semantically similar words or phrases. In this case, the recovered traceability links will not provide the benefits of using RTR and will be of little use. The challenges for automated RTR lie not only in the definition and in the description of the traceability links, but also in their maintenance when the artifacts change. Furthermore, when the artifacts are developed using different tools, the complexity of automatic RTR will grow. Due to the complexity of the deployed models and to the lack of implemented tools, the application of automatic RTR is limited and the relevant techniques are not widely adopted in industrial settings.

In this work, when dealing with the RTR problem, we focus on recovering traceability links between requirements and source code elements (e.g., method, class, or file) that implement them. Although, by doing so, we restricted the scope of the general RTR problem, the complexity, however, is still high enough as can be shown by the following formal presentation and analysis of the problem.

A requirement can be related to many source code elements, i.e., implemented by many source code elements. A source code element can be related to many requirements, i.e., used in the implementation of these requirements.

Let R be the set of requirements with $|R|= n$, i.e., n is the number of requirements.

Let C be the set of source code elements with $|C|= m$, i.e., m is the number of source code elements.

Let L be the list of candidate matches between requirements and source code elements produced by the RTR approach. L is a relation between R and C, and hence a subset of the Cartesian product between R and C, i.e., $L \subseteq R \times C$. Because $R \times C$ is a set consisting of n*m elements, there are $2^{n \times m}$ possible subsets of this set.

Obviously, the RTP problem has a large (exponential) number of potential solutions. Exploring this large search space using conventional methods/algorithms such as brute-force methods will be intractable. Such algorithms will be $\theta(2^{n \times m})$, i.e., will have exponential complexity.

To avoid the intractability of conventional methods, we will consider the RTR problem as an optimization problem and seek to solve it using genetic algorithms.

2.3 Knowledge incorporation into requirements traceability recovery

Knowledge can be incorporated into RTR in the form of user feedback, where the user in our case is the designer or developer. Feedback plays an important role in software development and software engineering in general and helps building better software [96–100]. There are many types of feedback. Feedback on priorities, for example, is related to the capabilities that need to be delivered and work organization. Feedback on design is related to properly capturing the requirements. Feedback on working software is related to building the right solution and planning necessary updates. Feedback on code is related to allowing a good code review workflow and experience. Some types of feedback enable collaboration between stakeholders and other types enable collaboration between developers.

As is the case for humans, where feedback clarifies understanding and helps seeing things in new ways, feedback can make RTR learn and correct the course of link detection, and hence work better and be more successful. Releasing feedback requests at specific and spaced out points in time, during the RTR search process carried out by the genetic algorithm, can give the algorithm interesting insight into designer/developer satisfaction about the link detection over time and guide the search process.

We use two types of feedback metrics, namely binary feedback and a more granulated feedback using number range ratings. In binary feedback, the designer/developer is asked to rate each detected link between a requirement and a source code element with either YES (1) or NO (0). YES means that designer/developer is satisfied with the correctness of the detected link. NO means that the link does not make sense and the designer/developer is not satisfied with its correctness.

In the more granulated feedback, the designer/developer is presented with a scale consisting of the number range [0, 0.3, 0.5, 0.8, 1], with the following meaning:

- 0: the proposed link does not make sense and the designer/developer is *not satisfied at all* with its correctness.
- 0.3: although the proposed link could make some sense, the designer/developer is *not satisfied* with its correctness.
- 0.5: the designer/developer is *neutral/undecided* concerning the correctness of the proposed link.
- 0.8: the proposed link seems to make sense and the designer/developer is *satisfied* with its correctness.

- 1: the proposed link makes sense and the designer/developer is *very satisfied* with its correctness.

2.4 Genetic algorithms and interactive genetic algorithms

Genetic algorithms [101–103] are heuristic methods that adapt principles of evolutionary biology to solve optimization problems. They mimic the paired biological processes of natural selection and sexual reproduction. They are universal tools that can deliver good results in a short period of time. They can therefore be applied when conventional methods, for complexity reasons, fail to solve optimization problems having a large number of potential solutions. When confronted with high dimensionality, genetic algorithms avoid the intractability of brute-force methods and are still able to explore a wide set of potential solutions. They do this by looking first for partial solutions to the problem. These are then used to form fuller solutions by defining discrete neighborhoods.

The basic idea in a genetic algorithm is to change and combine initial candidate solutions for the intended problem until a sufficiently good solution has been found. Figure 3 depicts the general form of a genetic algorithm.

An individual or chromosome of the population P corresponds to a candidate solution from the search space of the problem. The individual consists of genes and can be represented as a string of characters, a sequence of numbers, a sequence of variables, etc. The first step in the algorithm is to generate a starting population with random characteristics. This is usually done by choosing individuals randomly from the search space. In the next step, each individual in the population P is assigned a fitness value using the fitness function. The fitness function corresponds to the objective function of the optimization problem. The bigger the fitness value of an individual (compared to the fitness values of other individuals in P), the greater is its survival probability. In the following step, the algorithm uses the selection operator to select those individuals that make up the next generation of the population by taking into account the survival probabilities. Individuals that are selected several times should be renamed. The new generation contains as many individuals as the previous generation. After selection has ensured the survival of the fittest, the genetic operators mutation and crossover are applied in the following step with the aim of affecting the population variety and allowing to strive toward the achievement of optimal solutions by generating new and fitter individuals. The mutation operator changes randomly one or more genes of an individual. The mutation of each gene of each individual of the population happens according to a given mutation probability. The crossover operator combines two parent individuals and generates two new child individuals that will replace the parents in the population. Each of the child individuals inherits some features from the first parent and some from the second parent. Pairs of individuals selected from the population P are crossed over with a certain probability. Each individual is involved only once.

Input: optimization problem with search space S and fitness function f
Output: best individual from the final population (optimized solution)

```
Generate starting population P randomly
WHILE (termination criteria not reached)
{
        Calculate the fitness of the individuals in P
        Select according to fitness
        Apply genetic operators
}
```

Fig. 3 General form of a genetic algorithm

While the genetic algorithm is looping, changes occur and can produce new and better solutions (individuals). However, there is no warranty that the new solutions will always be better than the previous ones. This will be the case, for example, when after a while, a number of individuals prevail and no significant change can occur. The fitness of the individuals could also decrease through mutation or crossover. It could also be the case that the optimum has already been found and no further improvement is possible. For this reason, it is necessary to define a termination criterion that informs the algorithm when the individuals of the population represent a sufficiently good solution. The algorithm can stop, for example, after a certain number of iterations. It could also terminate when there is no improvement over the previous generation, or when the average improvement of the last generations has fallen below a threshold. When the termination criterion is met, the algorithm ends and returns the individual that has the best fitness in the population as the result.

Having humans in the loop of a genetic algorithm gave rise to advanced optimization methods called Interactive Genetic Algorithms (IGAs), which are gaining popularity in multiple fields and are effective in solving optimization problems with implicit or fuzzy indices. These methods allow users to introduce their preferences and knowledge during the algorithm's search process in the form of user feedback. Figure 4 depicts the IGA process. The feedback is usually subjective and unquantified. By observing the phenotypes of different individuals in a generation using a human–computer interface, the user can influence the fitness of the individuals. There were many successful applications of IGAs in different fields (e.g., [104–110]). Evaluating individuals and expressing their fitness through feedback is very important for the performance and applicability of IGAs in complicated optimization problems, because user fatigue resulting from frequent interaction with the user may impose restrictions on

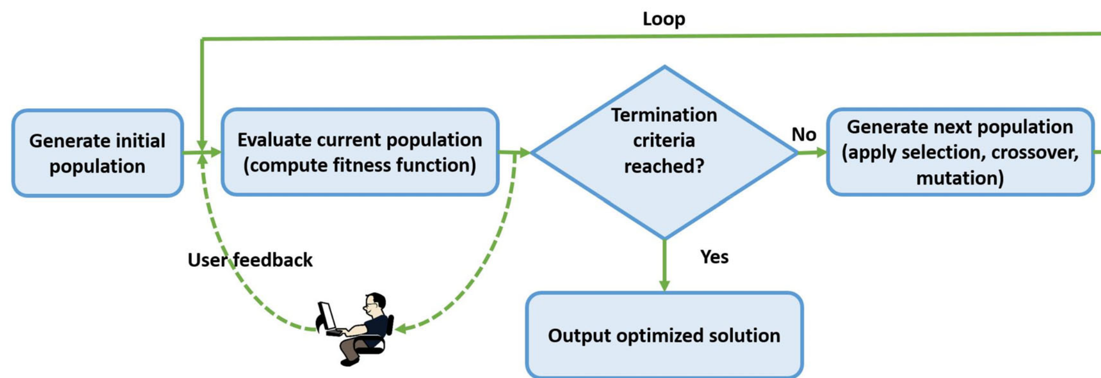


Fig. 4 The interactive genetic algorithm process

the population size and the number of evolutionary generations [111].

2.5 Motivation

In this subsection, we provide a motivating example that illustrates the proposed work.

Suppose a company wants to develop a new large software system and reuse as much as possible code from an existing system, the company could apply a typical similarity-based method. The output of this requirements traceability recovery process would consist of a list of candidate links. This list contains pairs of software artifacts (requirement, source code element) ordered on the basis of their textual similarity. The method works, but the accuracy of the recovered links may be unsatisfactory, i.e., the recall and precision values are low.

These results provide a motivation to extend the similarity-based method. A heuristic search algorithm can be an ideal method that goes through requirements and source code element lists to automatically identify correct links. This can be done by a GA-based method that relies not only on textual similarity between the requirements and source code elements but also takes into account some other properties of the source code elements. For example, given in input a list of requirements (R1, R2, R3, …) and a list of source code elements (SCE1, SCE2, SCE3, …), the output of this process can be as follows:

Requirement	Source code element
R2	SCE1
R9	SCE21
R1	SCE20
R19	SCE32
R11	SCE25
R7	SCE26
R4	SCE18
R3	SCE20
R4	SCE1

The method works, and the accuracy of the recovered links improves, i.e., the recall and precision values are higher than the previous ones, but improvement is still possible. Why wait until the GA converges and returns the links? Why not support the system during the search process through user (designer/developer) feedback? Periodically, after a certain time, the system asks the user for feedback (binary feedback or granulated feedback) as follows:

Requirement	Source code element	Binary feedback	OR	Granulated feedback
R2	SCE1	1		0.75
R9	SCE21	1		1
R1	SCE20	1		0.5
R19	SCE32	1		0.25
R11	SCE25	1		0.75
R7	SCE26	0		0.25
R4	SCE18	1		1
R3	SCE20	0		0
R4	SCE1	0		0.25

The approach works, and the accuracy of the recovered traceability links between requirements and source code elements improves significantly.

3 Related work

Requirements traceability recovery for the assistance in software reuse and other software engineering tasks is based in many cases on information retrieval techniques [112–114], such as the probabilistic model [21, 23], the vector space model [22, 115, 116], and extensions of these models like latent semantic indexing [67, 117, 118]. These methods are usually similarity-based methods. They treat both requirements and source code as plain text and recover links by computing the similarity between the representations of

requirements and source code. It is usually assumed that most of the identifiers in the source code are named with meaningful words. If, however, there are traceability links that do not depend on the textual similarity between the representations, these methods will not work well. Furthermore, information retrieval methods applied to recover traceability links usually use parameters that were tuned for applications that retrieve natural text. However, the textual data present in software artifacts are not necessarily the same as the textual data present in documents containing natural text. It is also not clear if the data in one software system have the same characteristics as in another one. To cover the weaknesses of these methods and improve the quality and accuracy of recovery, many approaches have been proposed.

In [119], for example, the authors proposed to improve latent semantic indexing using hierarchical structure enhancement, similarity thesaurus, identifier classifying, and source code clustering.

In [120], the authors proposed to extend the similarity-based methods by applying natural language processing and document retrieval to the configuration management log.

In [33], the focus was on improving the traceability links generated by information retrieval methods by considering the vocabulary base used for performing traceability and also by suggesting the use of secondary measures for the evaluation and assessment of traceability mappings.

In [121], improving information retrieval-based traceability recovery is attempted using a smoothing filter as a complement and alternative to stop word removal. The filter removes terms that do not allow discriminating artifacts, i.e., terms that do not help to characterize the semantics of the artifact.

In [17], the authors proposed to improve the accuracy of information retrieval-based traceability recovery by combining the results of mining software repositories with information retrieval techniques. They present Trustrace, an approach for recovering traceability links between requirements and source code. Heterogeneous sources of information are used to discard/re-rank traceability links that are generated by information retrieval techniques.

In [32], a combination of textual and structural information is suggested for the recovery of traceability links between artifacts in situations where the artifacts describe different tasks and may use different terminology. Structural analysis performs the examination of control and data flow among the artifacts and helps to alleviate this particular problem and to allow the recovery of useful traceability links.

In [122], the author suggested to enhance information retrieval-based traceability recovery by providing support for the storage and incorporation of previous user feedback across several retrieval sessions. This helps in dealing with artifact evolution by avoiding the recalculation of link candidates from scratch when the artifact is changed and, hence, releasing the user from making decisions he already made in the past, repeatedly.

In [123], the authors aimed at reducing the number of false positives during traceability recovery, i.e., avoiding links between pairs of artifacts having high textual similarity but that are not related to each other. They proposed an adaptive version of relevance feedback. The decision whether to apply feedback and how is based on the characteristics of the artifacts and on the characteristics of the previously classified links.

In [124], it is suggested to improve information retrieval-based traceability recovery by introducing a concept-based semantic representation to provide a semantic and unified description of system artifacts. Instead of adopting an informal (text-based) representation for the artifacts, a formal (concept-based) representation is adopted. A knowledge-based layer identifying terms and entities is used to bridge the gap between natural language and a domain-specific vocabulary (concepts).

In [125], and in an attempt to extend approaches for semi-automatic link recovery across requirements and source code in which textual analysis and information retrieval techniques are the baseline, a method that further enables the automatic generation of links is targeted. In this method, RTR is investigated as a combinational problem using an optimization approach aiming at automating the link recovery process. The RTR problem is studied as a big search space consisting of pairs of requirements and source code elements that are matched to each other using the artificial bee colony (ABC) algorithm.

In [126], the authors argue that reusing knowledge from existing projects and traceability between corresponding artifacts are important steps toward automatic software and system development. They list barriers that are encountered in industry with software artifact reuse and traceability and suggest ideas to overcome these barriers.

In [127], the authors introduce a query quality prediction approach for software artifact retrieval by adapting natural language-inspired solutions for use on software data. They apply the approach in the context of traceability link recovery where the queries represent software artifacts. They report that the approach can identify artifacts that are hard to trace to other artifacts and may therefore have a low intrinsic quality for text retrieval-based traceability link recovery.

In this work, we propose to consider requirements traceability recovery for the purpose of software reuse as an optimization problem. We extend information retrieval-based traceability recovery by taking into account some properties of the artifacts that are related to the history of reuse. We also integrate user feedback into the search process.

4 Modeling the RTR problem using genetic algorithms

In this section, we introduce the basic GA and IGA concepts, namely the representation of individuals, the genetic operators, the evaluation of individuals (fitness function), and the collection and integration of user (designer/developer) feedback. After that, we present a high-level pseudo-code of the IGA that is adapted to the RTR problem.

4.1 Representation of individuals

To apply the GA and IGA, we represented an individual (i.e., a candidate solution) as a list in which each element (i.e., gene) is a pair (R, E). R is a requirement description selected from the set of requirements at hand. E is a source code element (e.g., class, method, file, etc.) selected from the source code of the software system under analysis. Figure 5 shows an example of an individual with three elements.

4.2 Genetic operators

Genetic operators guide the genetic algorithm toward a solution to a given problem. Selection, crossover, and mutation are the main types of genetic operators that influence the success of the algorithm.

4.2.1 Selection

A multitude of selection methods were proposed in the literature aiming at selecting the best individuals that will take part of the next generation (e.g., roulette wheel selection [128], Boltzmann selection [129], tournament selection [130], rank selection [131], steady state selection [132], etc.). Many researchers [132–134] argue that, in certain cases, the choice of the adequate selection method can help avoid bias toward highly fitted individuals.

The rank selection technique, for example, is recommended especially when some individual fitness function values are very high compared to others, in order to avoid that these individuals dominate the selection process (as it is the case with roulette wheel selection, for example).

In this paper, we run the GA and IGA using the rank selection technique to generate a new population, i.e., the set of individuals that will undergo the crossover and mutation operators, from the current population. Rank selection [131] consists of ranking the current population. The individuals in the population are sorted in order of highest fitness to lowest fitness with the highest being awarded a rank of N (population size) and the lowest being awarded the rank one. The associated rank is then used as fitness value for each individual. The higher the rank, the higher the fitness.

Requirement	Code Element
R6	E1
R2	E2
R7	E5

Fig. 5 Representation of an individual

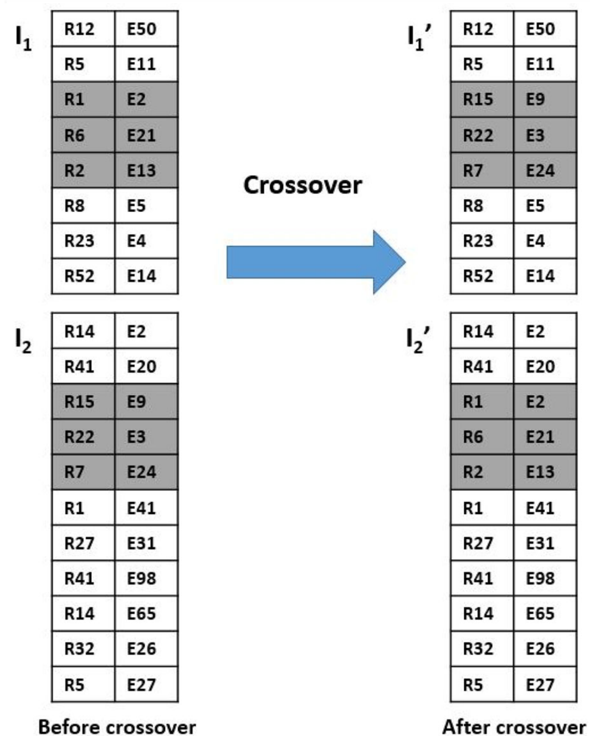


Fig. 6 Crossover operator

4.2.2 Crossover

There are many types of crossover in the literature (single point, two-point, uniform, arithmetic, etc.). In this paper, we used a double, random, cut-point crossover. Given two selected individuals (i.e., parents) I_1 and I_2 based on rank selection and a certain probability parameter, the crossover operator allows creating two children I_1' and I_2' from the two selected parents I_1 and I_2. A two-point crossover (Fig. 6) consists of randomly selecting two points from the parents I_1 and I_2, then everything between the two points is swapped between the parents, creating two children I_1' and I_2'.

4.2.3 Mutation

Given a selected individual, the mutation operator randomly selects one or more elements (i.e., genes) from the list corresponding to the selected individual (solution) I. Then, these

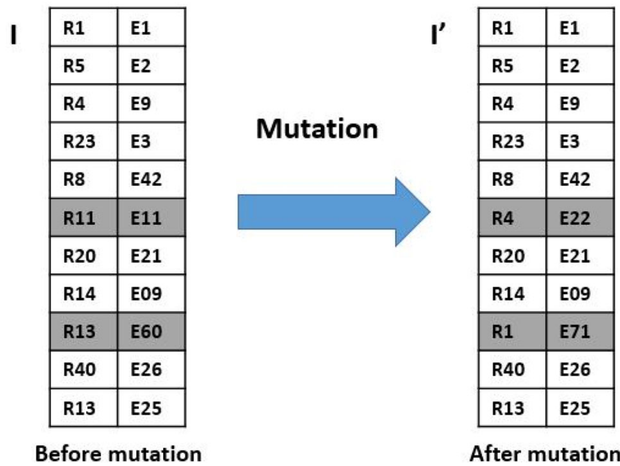


Fig. 7 Mutation operator

elements are replaced by elements (pairs having the form (R, E)) chosen randomly from the base of examples to form the new individual I'. Figure 7 shows the effect of a mutation operation that replaces the two pairs (R11, E11) and (R13, E60) by the pairs (R4, E22) and (R1, E71), respectively.

4.3 Evaluation of individuals (fitness function)

The evaluation of individuals is based on two factors: (1) semantic similarity between requirements and source code elements and (2) some statistics from the reuse history. To measure the semantic similarity, we used the cosine of the angle between a vector that represents the requirement and a vector that represents the code element. The statistics from the reuse history include two measures: (i) Recency of reuse, and (ii) Frequency of reuse.

4.3.1 Semantic similarity

To define the semantic similarity, we assume that developers use consistent naming in various artifacts and, hence, artifacts (requirements and code elements) having a high textual similarity are good candidates to establish links between them. To establish a link between a requirement and a code element, we compute the semantic similarity between the requirement text and the code element text (i.e., names of classes, methods, fields, variables, parameters, types, etc.). We use the cosine similarity measure known in the field of Information Retrieval (IR) [135]. Requirements and code elements are considered as documents.

The documents are represented as vectors in a v-dimensional space where v is the number of the different terms in the vocabulary (all different terms appearing in the documents). The vector for each document is obtained by assigning a weight to each dimension (representing a specific term) of the term space. The weight w_{ij} corresponding to the term k_i and document d_j is computed using the Term Frequency-Inverse Document Frequency (TF-IDF) method known in the field of IR [136] as shown in Eq. (1). tf_{ij} is the term frequency factor. idf_i is the inverse document frequency factor. $freq_{ij}$ is the raw frequency of term k_i in document d_j. $\max_l \text{freq}_{lj}$ is the maximum term frequency in document d_j. N is the total number of documents and n_i is the document frequency of k_i, i.e., the number of documents in which the term k_i appears, where $i = 1, 2, \ldots, v$ and $j = 1, 2, \ldots,$ N.

$$w_{ij} = tf_{ij} \cdot idf_i = \left(\frac{\text{freq}_{ij}}{\max_l \text{freq}_{lj}} \right) \cdot \log\left(\frac{N}{n_i} \right) \tag{1}$$

The similarity between a requirement R and a code element E is computed by determining the cosine of the angle between the vector $\overrightarrow{R}$ representing the requirement R and the vector $\overrightarrow{E}$ representing the code element E, i.e., inner product normalized by the vector lengths, as shown in Eq. (2):

$$CosSim\left(\vec{R}, \vec{E}\right) = \frac{\vec{R} \cdot \vec{E}}{\left|\vec{R}\right| \cdot \left|\vec{E}\right|} \tag{2}$$

SemSim, the function that computes the semantic similarity value for an individual I, is defined as the average of the semantic similarity values for the pair (R_i, E_i) over all pairs in the individual I as shown by Eq. (3), where m is the number of pairs in the individual I (i.e., individual size).

$$SemSim(I) = \frac{1}{m} \sum_{i=1}^{m} CosSim\left(\vec{R}_i, \vec{E}_i\right) \tag{3}$$

4.3.2 Reuse history

From the reuse history, we use the following two types of statistics: the recency of reuse of code elements, and the frequency of reuse of code elements.

Recency of Reuse Measure (RR): The RR measure is computed based on information extracted from the history of reuse of the code elements of the software system. The intuition behind introducing the RR measure is that code elements (classes, methods, etc.) that were reused more recently than others are more likely to be reused now, i.e., are related to the new requirement at hand. The RR measure for a given code element is obtained by looking to time at which the code element in the software system has been reused. The RR measure for a code element E is given by Eq. (4), where LTR stands for "Last Time of Reuse." The value of the function *RecOfReuse* that computes the recency of reuse for an individual I is computed as the average of the recency of reuse value for the code element E_i over all code elements in the individual I. This average is then normalized by the

current time to ensure that the value is in the interval [0, 1]. This is shown by Eq. (5), where m is the number of code elements, i.e., the number of pairs, in the individual I (i.e., size of the individual).

$$RR(E) = LTR_E \quad (4)$$

$$RecOfReuse(I) = \frac{\frac{\sum_{i=1}^{m} RR(E_i)}{m}}{\text{Currenttime}} \quad (5)$$

Frequency of Reuse Measure (FR): The FR measure is computed based on information extracted from the history of reuse of the code elements of the software system. The intuition behind introducing the FR measure is that code elements (classes, methods, etc.) that are reused more frequently than others are more likely to be reused now, i.e., are related to the new requirement at hand. The FR measure for a given code element is obtained by looking to the number of times the code element in the software system has been reused. The FR measure for a code element is given by Eq. (6), where NR stands for "Number of Reuses." The value of the function *FreqOfReuse* that computes the frequency of reuse for an individual I is computed as the average of the frequency of reuse value for the code element E_i over all code elements in the individual I. This average is then normalized by the maximum number of reuses (over all code elements in the individual) to ensure that the value is in the interval [0, 1]. This is depicted by Eq. (7), where m is the number of code elements, i.e., the number of pairs, in the individual I (i.e., size of the individual).

$$FR(E) = NR_E \quad (6)$$

$$\text{Freq of Reuse}(I) = \frac{\frac{\sum_{i=1}^{m} FR(E_i)}{m}}{\max_l FR(E_l)} \quad (7)$$

4.3.3 Fitness function

The GA/IGA is a mono-objective optimization technique. To apply the GA/IGA, the evaluation of an individual should be formalized as a mathematical function called "fitness function." The fitness function F1 used in this work combines the three functions described above, namely *SemSim*, *RecOfReuse* and *FreqOfReuse*. In the fitness function formula depicted in Eq. (8), the weights assigned to *SemSim*, *RecOfReuse*, and *FreqOfReuse* were set to 50%, 25%, and 25%, respectively. The choice of these weights was through trial and error.

$$F1(I) = \frac{SemSim(I) + \frac{RecOfReuse(I)+FreqOfReuse(I)}{2}}{2} \quad (8)$$

4.4 Collection and integration of user feedback

RTR is a context-sensitive operation. The person who is responsible for this operation is either the designer or the developer or both. The designer/developer can judge the correctness of a traceability link between a requirement and a source code element proposed by an optimization technique. The integration of the domain-specific knowledge in the RTR system is done through interaction with the designer/developer by feedback collection.

In general, when collecting feedback, the basic principle as discussed in [97], is that, during an experiment, subjects are asked for feedback at different times. These requests for feedback are called probes. The frequency of probing and the time allowed for feedback are two parameters to consider during feedback collection. There are many strategies that can be used to decide about the frequency of probing. Probing can happen at random points, at regular intervals, designer/developer driven (i.e., whenever the designer/developer thinks it is appropriate), event-driven (e.g., the best fitness value reaches certain thresholds), or according to a combination of these strategies.

In our case, as for the frequency of probing, the IGA asks the designer/developer for feedback at specific and spaced out points in time, during the search process, i.e., repeatedly after a fixed number of iterations. The time allowed for the designer to provide feedback is not limited.

During the execution of the IGA, the designer/developer is asked to rate the traceability links between requirements and code elements detected by the algorithm so far by providing feedback. This is done each time a defined number of iterations is performed. As mentioned earlier, we use two types of feedback metrics, namely binary feedback and a more granulated feedback using ratings in the number range [0, 0.3, 0.5, 0.8, 1].

Feedback is integrated into the IGA through the new fitness function $F2$ given by Eq. (9).

$$F2(I) = \frac{F1(I) + \frac{\sum_{i=1}^{m} Feedback(R_i,E_i)}{m}}{2} \quad (9)$$

In Eq. (9), $Feedback(R_i,E_i)$ represents the feedback provided by the designer/developer for the traceability link between requirement R_i and source code element E_i. The feedback value for an individual I is computed as the average of feedback value for the pair (R_i,E_i) over all pairs (links) in the individual I. m is the number of pairs, in the individual I (i.e., size of the individual). The feedback value for an individual I is a value in the interval [0, 1]. The fitness function $F2$ averages the value of fitness function $F1$ (which is based on semantic similarity and history of reuse) with the feedback provided by the designer/developer. The intuition behind this choice was to ovoid being led astray by focusing

Fig. 8 Algorithm for the IGA adaptation to the RTR problem

```
Input: Set of requirements; Set of code elements; Reuse history; Percentage (P%); MaxNbrIterations; NbrIterations;
       NbrInteractions
Output: Set of traceability links between requirements and code elements.

01: for i = 1 … NbrInteractions do
02:     Evolve GA for NbrIterations
03:     Select P% of best solutions (individuals) from the current population
04:     for-each selected solution do
05:         Ask the designer to provide feedback about each traceability link in the selected solution
06:         Update the fitness function value of the selected solution to integrate the feedback
07:     end for-each
08:     Create a new GA population using the updated solutions
09: end for
10: Continue (non-interactive) GA evolution until it converges or it reaches MaxNbrIterations
```

on the wrong thing, i.e., not focusing too much on semantic similarity and history of reuse, nor on feedback.

4.5 Interactive genetic algorithm adaptation

We aim at detecting traceability links between requirements and source code elements by the means of a heuristic search technique that integrates semantic similarity, reuse history, and designer/developer's feedback. The algorithm in Fig. 8 shows the pseudo-code of the IGA adaptation to the RTR problem.

The algorithm takes as input a set of requirements, a set of source code elements, statistics from reuse history. The algorithm takes also as input a percentage value corresponding to the percentage of a population of solutions (individuals) that the designer/developer is willing to provide feedback for, the maximum number of iterations for the algorithm, and the number of interactions with the designer/developer during which feedback is provided. "NbrIterations" is the number of iterations the algorithm is allowed each time to run before asking for feedback. Initially (line 2), the algorithm runs for a certain number of iterations (i.e., usually the maximum number of iterations divided by the number of interactions). Then, a percentage of the solutions from the current population is selected (line 3). In lines 4–7, we receive the designer/developer's feedback for each traceability link in each selected solution, then, we update the fitness function value of each selected solution. We then generate a new population of individuals (line 8) by applying the crossover operator and mutation using a probability score in order to ensure diversity. This step produces the population for the next generation. The algorithm terminates when the maximum iteration number is reached and returns the best solution.

5 Empirical study

In this section, we describe the definition, design, and setting of the experiments, following the general guidelines by Wohlin et al. [137].

5.1 Experiment definition, design and context

The goal of the experiments was to evaluate the proposed RTR approach. As mentioned earlier, we will evaluate the approach using three object-oriented open-source projects. In a first set of experiments, we perform the evaluation by running a GA to conduct RTR automatically based only on semantic similarity and reuse history. In a second set of experiments, we incorporate additionally user (designer/developer) knowledge in the form of feedback using an IGA. We conduct this evaluation using two types of feedback metrics, namely binary feedback and a more granulated feedback using ratings in the number range [0, 0.3, 0.5, 0.8, 1].

We aim at supporting the developer/designer who has a set of requirements and is looking for code elements to reuse. These code elements should be related to the requirements at hand. We want to find out whether the presented approach could help in dealing with the recovery of traceability links between the requirements and the code elements of a software system. We want to evaluate the ability of the approach to generate correct traceability links by assessing its performance and its stability, i.e., its ability to allow a stable performance during different executions. We will also compare the performances of the GA, the IGA, and another approach proposed in the literature. We will determine the precision and recall of the approach by applying it on a set of existing projects for which we have information about the relationships (traceability links) between requirements and code elements. We will run the approach multiple times (31 executions for each project) and observe its behavior in terms of precision and recall scores.

5.2 Experimental settings

Parameter tuning was performed by trial and error by executing several tests. The following settings were found to work well. The "Maximum number of iterations" (stopping criterion) parameter was set to 10,000. The "Maximum size of the population" parameter was set to 40. The "Crossover proba-

bility" parameter was set to 90%. The "mutation probability" parameter was set to 10%. This relatively high mutation rate allows to diversify the population continuously and discourages the occurrence of premature convergence.

To run the experiments, we used a HP Proliant DL580 (Gen8) server (Two processor Intel Xeon E7-4820v2 at 2.0 GHz with 8-core, Cache Memory 20 MB, 64 GB of RAM). The execution time of the approach depends on the number of requirements and code elements, and on the length of their texts. In the GA case, the execution time of the algorithm, i.e., the time needed for performing 10,000 iterations, was less than 5 min, which is a good indicator for the scalability of the approach. In The IGA case, the execution time was mainly dependent on the time needed by the developer/designer to enter the feedback.

5.3 Description of the datasets

For the evaluation of the approach, we used three medium-sized open-source projects that are widely used in traceability recovery evaluation [18, 19, 65, 67, 138–141]:

- LEDA (Library of Efficient Datatypes and Algorithms) [142]: A freely available C++ library of foundation classes developed and distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany. We used the release 3.4.
- Albergate: A Hotel management system developed in Java, according to a waterfall process, by a team of final year students at the University of Verona (Italy). Available at http://coest.org/ (CoEST).
- eTour: A Tour guide system. Available at http://coest.org/ (CoEST).

For all three projects LEDA, Albergate, and eTour, we have the requirements, the source code, and the traceability links between requirements and source code elements. We use these projects for the evaluation of our approach. We launch our approach (i.e., the 3 variants of the approach, each variant in a separate set of experiments) on these systems in order to evaluate its ability to detect the traceability links.

The Library of Efficient Data types and Algorithms (LEDA) is a software library providing C+ + implementations of a broad variety of algorithms for graph theory and computational geometry. It was originally developed by the Max Planck Institute for Informatics, Saarbrücken, Germany. Now it is further developed and distributed by the Algorithmic Solutions Software GmbH (https://www.algorithmic-solutions.com/index.php/products/leda-free-edition). We used an older version (release 3.4). The data sets for the Albergate and eTour systems were downloaded from http://coest.org/ (CoEST). The CoEST (Center of Excellence for Software & Systems Traceability) is a website created to provide data sets for RTR investigations.

Table 1 Details of the datasets

Project name	KLOC	# Classes	# Methods	# Requirements
Leda	95	208	283	88
Albergate	20	95	119	17
eTour	65	116	212	58

We selected the systems LEDA, Albergate and eTour because they include the requirements document, source code divided by classes, and traceability links from the requirements document to the source code elements of each system.

When the approach is to be used in a real-world application, a reuse environment that stores information about the reuse history of the individual code elements will provide the recency of reuse and frequency of reuse data for the source code elements. In our case, when evaluating our approach, no reuse history data were available. For the purpose of the evaluation, we assumed values for the recency of reuse and frequency of reuse of the source code elements, based on our estimation (subjective estimation based on the generality of the source code elements) of the reuse probabilities of the source code elements.

The details of these datasets are shown in Table 1.

5.4 Experimental variables

In the context of the proposed experiment, the independent variable was the RTR method used. We used the following four treatments: GA (proposed in this work), IGA with binary feedback (proposed in this work), IGA with granulated feedback (proposed in this work), and IR-based method (from related work).

The dependent variable, the outcome observed in the study, was the accuracy of the tracing result. We use two widely accepted information retrieval metrics, namely recall and precision [143] to measure the accuracy and access the approach. Recall is the ratio of the number of correct traceability links (between requirements and source code elements) retrieved by the approach over the total number of correct traceability links (Eq. 10). Precision is the ratio of the number of correct traceability links retrieved by the approach over the total number of traceability links retrieved by the approach (Eq. 11). In general, precision is a measure of quality. When the approach returns more correct links than incorrect ones, precision is high. Recall is a measure of quantity. When the approach returns most of the correct links, recall is high.

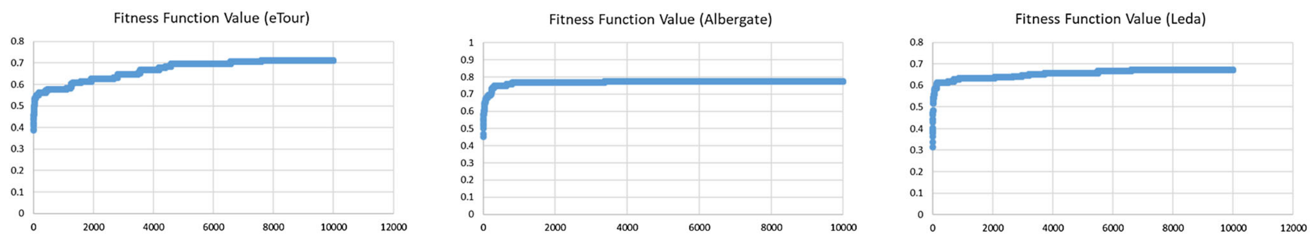


Fig. 9 Evolution of the fitness functions for the three projects eTour, Albergate, and Leda

Fig. 10 Recall and precision results for 31 executions of the GA on the eTour project

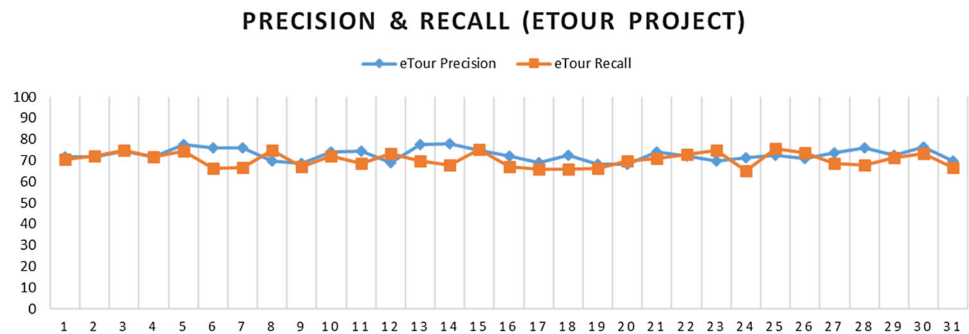


Fig. 11 Recall and precision results for 31 executions of the GA on the Albergate project

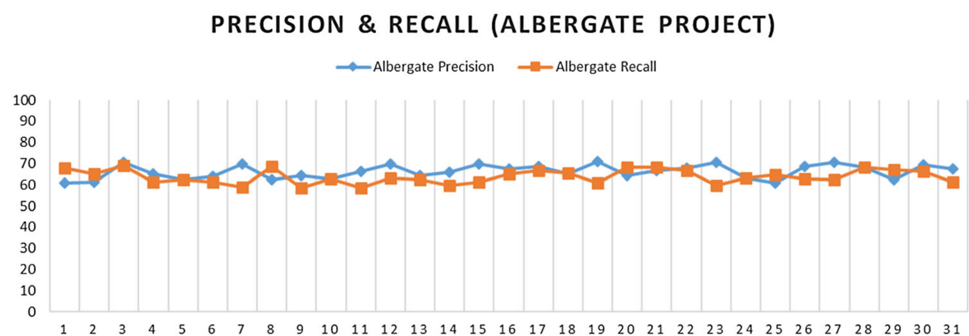
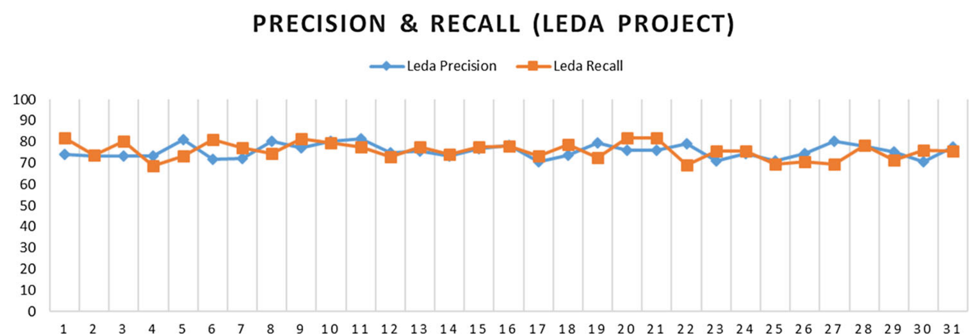


Fig. 12 Recall and precision results for 31 executions of the GA on the Leda project



Precision and recall require the use of knowledge or inference as to the correct answers. As mentioned earlier, for all three systems that we used for the evaluation of our approach, namely LEDA, Albergate, and eTour, we have the requirements, the source code, and the traceability links between requirements and source code elements.

$$\text{RTR_recall} = \frac{|\text{correct retrieved}|}{|\text{correct}|} \tag{10}$$

$$\text{RTR_precision} = \frac{|\text{correct retrieved}|}{|\text{retrieved}|} \tag{11}$$

5.5 Results and discussion

In a first set of experiments, we performed the evaluation by running a GA to conduct RTR automatically based only on semantic similarity and reuse history on the projects shown in Table 1. Figure 9 shows the evolution of the fitness functions for the three projects eTour, Albergate, and Leda. Figures 10, 11, and 12 depict the precision and recall values (y-axis) obtained for 31 executions (x-axis) for eTour, Albergate, and Leda, respectively. The average (precision, recall) values

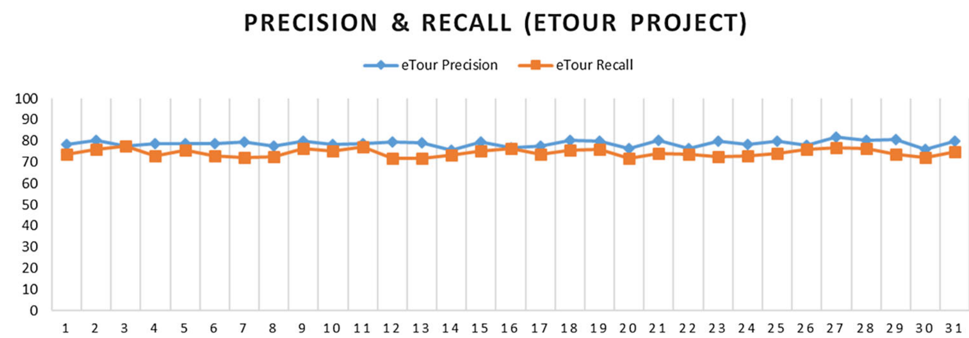


Fig. 13 Recall and precision results for 31 executions of the IGA with binary feedback on the eTour project

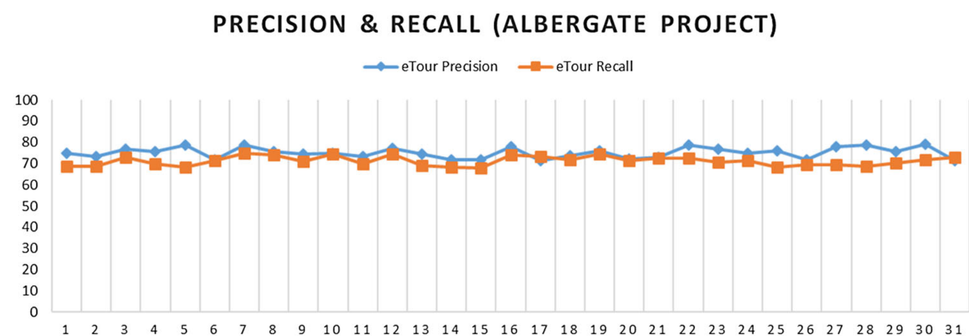


Fig. 14 Recall and precision results for 31 executions of the IGA with binary feedback on the Albergate project

(averaged over the 31 executions) were as follows: (72.71%, 70.35%) for eTour, (66.16%, 63.78%) for Albergate, and (75.60%, 75.70%) for Leda.

Despite the randomness effect that is present in any meta-heuristic technique application, the approach looks to be stable enough as there are no noticeable fluctuations in the values of precision and recall during different executions of the approach for all three systems. These results are also promising with regard to performance expressed by precision and recall. For example, the average (precision, recall) values outperform those values reported in [138] when using the vector space model of information retrieval on the eTour project when they got (17%, 47%). The average (precision, recall) values also outperform those values reported in [65] when using the probabilistic information retrieval model on the Albergate project when they got (48.33%, 50%). The average (precision, recall) values also outperform those values reported in [140] when using the probabilistic information retrieval model on the Leda project when they got (25%, 53.06%).

It is noteworthy that the average (precision, recall) values for the Albergate project are relatively smaller than those values achieved by the GA approach for the Leda and eTour projects. This is probably related to the size of the Albergate project that has a smaller number of requirements and code elements. Heuristic search techniques usually perform better, i.e., produce better precision and recall scores, when the number of examples increases.

In a second set of experiments, we incorporated additionally user (designer/developer) knowledge in the form of feedback using an IGA. We conducted this evaluation using two types of feedback metrics, namely binary feedback and a more granulated feedback using ratings in the number range [0, 0.3, 0.5, 0.8, 1]. The number of interactions with the user was set to 10.

IGA requires feedback from humans. When the approach is to be used in a real-world application, experts, i.e., designers and developers of the software systems to be developed and the systems to be reused, and other experts in software engineering would provide such feedback. In our case, and for the purpose of evaluating the approach, providing the feedback is a relatively easy task, because we, the authors, already know the requirements, the source code, and the traceability links between requirements and source code elements. Hence, providing the feedback is straightforward and there is no risk of bias or lack of independent expertise.

Figures 13, 14, and 15 depict the precision and recall values (y-axis) obtained for 31 executions (x-axis) of the IGA with binary feedback for eTour, Albergate, and Leda, respectively. The average (precision, recall) values (averaged over the 31 executions) were as follows: (78.75%, 74.29%) for eTour, (75.08%, 71.11%) for Albergate, and (81.35%, 77.87%) for Leda.

Again, despite the randomness effect that is present in any meta-heuristic technique application, the approach looks to be stable enough as there are no noticeable fluctuations in the values of precision and recall during different executions of the approach for all three systems (see Figs. 13, 14, and 15). These IGA results with binary feedback are also promising with regard to performance expressed by precision and recall

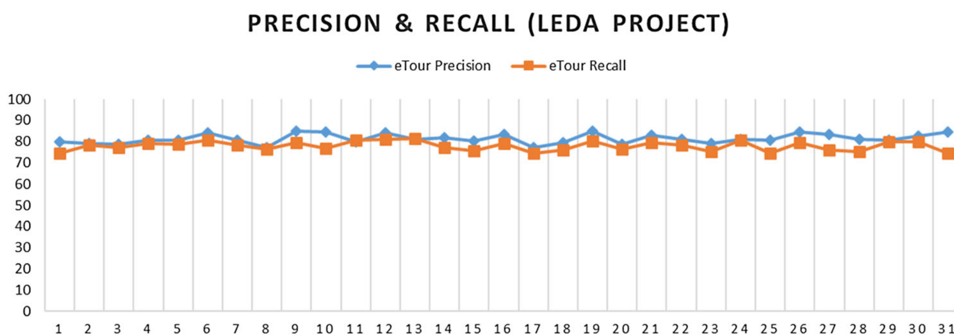


Fig. 15 Recall and precision results for 31 executions of the IGA with binary feedback on the Leda project

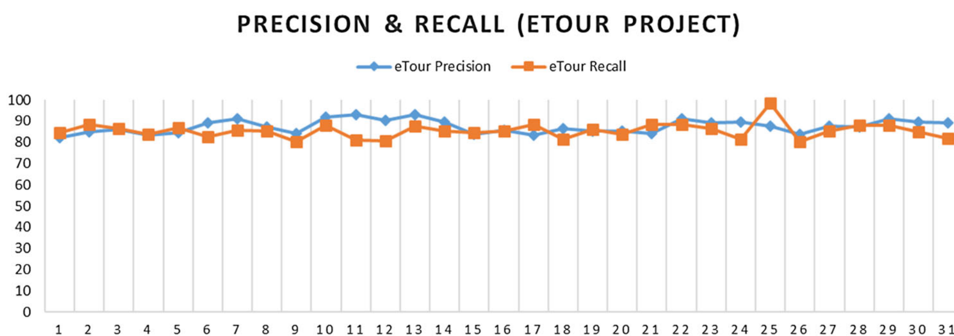


Fig. 16 Recall and precision results for 31 executions of the IGA with granulated feedback on the eTour project

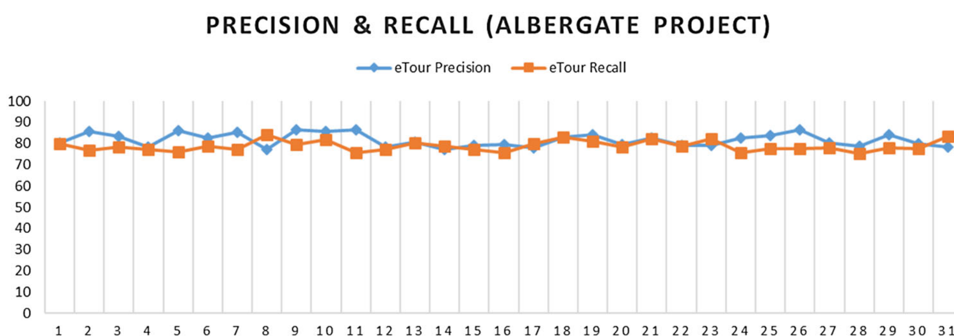


Fig. 17 Recall and precision results for 31 executions of the IGA with granulated feedback on the Albergate project

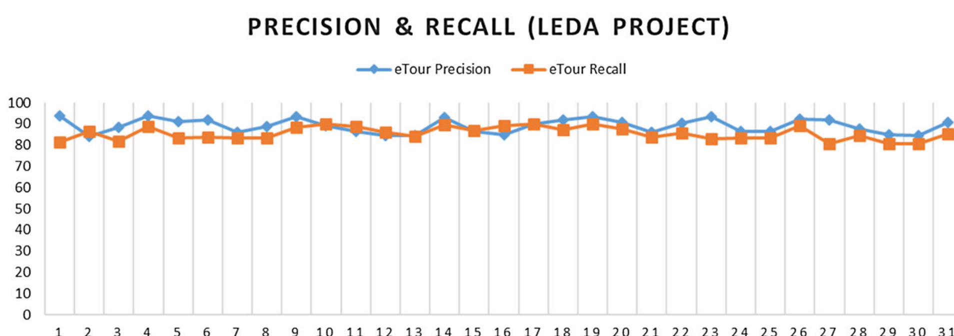


Fig. 18 Recall and precision results for 31 executions of the IGA with granulated feedback on the Leda project

as we notice a considerable improvement against the GA in both precision and recall levels for all three projects.

Figures 16, 17, and 18 depict the precision and recall values (y-axis) obtained for 31 executions (x-axis) of the IGA with granulated feedback for eTour, Albergate, and Leda, respectively. The average (precision, recall) values (averaged over the 31 executions) were as follows: (87.40%, 85.37%) for eTour, (81.63%, 78.74%) for Albergate, and (89.06%, 85.47%) for Leda.

Again, despite the randomness effect that is present in any meta-heuristic technique application, the approach looks to be stable enough as there are no noticeable fluctuations in the values of precision and recall during different executions of the approach for all three systems (see Figs. 16, 17, and 18). These IGA results with granulated feedback are also very promising with regard to performance expressed by precision and recall. There is a noticeably considerable improvement

Table 2 Summary of the results of the different approaches

Project	GA		IGA binary feedback		IGA granulated feedback		IR-based approaches	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Leda	75.6	75.7	81.35	77.87	89.06	85.47	25	53.06
Albergate	66.16	63.78	75.07	71.11	81.63	78.74	48.33	50
eTour	72.17	70.35	78.75	74.29	87.4	85.37	17	47

against the GA and the IGA with binary feedback in both precision and recall levels for all three projects.

Table 2 provides a summary of the presented results for the GA, IGA with binary feedback, IGA with granulated feedback, and IR-based approaches from related work (results for Leda from [140], results for Albergate from [65], results for eTour from [138]).

The results of the IGA-based approach with granulated feedback—measured in terms of two well known information retrieval metrics, namely precision and recall—are satisfactory and very encouraging and this enforces our belief that IGA-based RTR can play a useful role in software reuse.

5.6 Threats to validity

In our approach, we rely partly on computing the similarity between the representations of requirements and source code by treating both requirements and source code as plain text and assuming that source code identifiers (functions, variables, types, classes, and methods) are named with meaningful words. If this is not the case, the performance can decrease. However, we believe that professional programmers will ensure that they capture the application-domain knowledge that they process by the mnemonics for identifiers when they write the code. For traceability links that do not depend on the textual similarity between the representations, we rely on the reuse history and on the incorporated designer/developer knowledge to support the recovery.

Another problem that could influence the effectivity of the approach is user fatigue resulting from the user's evaluation of the individuals. In order to alleviate user fatigue, the user is just asked to provide feedback in certain points in time.

Additionally, the use of precision and recall for evaluating the effectiveness of the approach may not cover all its strengths and weaknesses. To assess other dimensions, like the usefulness of the recovered links for reuse, it would be interesting to consider additional measures.

For example, Sundaram et al. [33] used a measure called "selectivity" that can be used instead of precision in order to determine whether the candidate lists of traceability links returned by the approach are of acceptable sizes. They define selectivity as $|L|/(n*m)$ where L is the list of candidate matches between requirements and source code elements produced by the RTR approach, n is the number of requirements, and m is the number of source code elements. In general, when an analyst has to perform RTR manually, each requirement has to be compared to each source code element, which means that there are n*m potential candidate matches to be checked. Selectivity measures the savings incurred by the analyst when manually going through the list L generated by the automated RTR approach rather than manually comparing each (requirement, source code element) pair. The smaller the selectivity, the better the savings for the analyst. The authors argue that selectivity is not an exact measure of effort savings, because it assumes that the analyst will be correcting only errors of commission found in the list of candidate matches. Selectivity needs therefore to be considered in concert with recall. The higher the recall, the fewer errors of omission the analyst needs to fix, the better selectivity approximates effort savings. The authors came to the conclusion that in certain cases, secondary measures, like selectivity, significantly affect the analyst's perception of the quality of the results. This means that, in some cases, the assessment of the results of a trace given primary measures, like recall and precision, can be different from the assessment of the results using secondary measures, like selectivity.

6 Conclusions

In this paper, to support software reuse, we proposed a novel search-based RTR approach using genetic algorithms, that relies not only on semantic similarity between software artifacts, but also takes into account the history of reuse of the artifacts, and incorporates knowledge into RTR in the form of user (designer/developer) feedback.

We implemented the approach as a plugin integrated within the Eclipse platform and we performed multiple executions of the approach on three open-source projects. The results of the experiments show that the approach is stable and is performing well. The IGA with granulated feedback delivered the best results and looks to be promising.

While the results of the approach are very promising in terms of precision and recall, we plan to extend it in two different ways: (1) expand our study to include additional

datasets and (2) do more experimentations and analyses of feedback use.

References

1. Mäder P, Egyed A (2012) Assessing the effect of requirements traceability for software maintenance. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE
2. Adithya V, Deepak G (2021) OntoReq: an ontology focused collective knowledge approach for requirement traceability modelling. European, Asian, Middle Eastern, North African conference on management & information systems. Springer, Berlin
3. Falessi D et al (2017) Estimating the number of remaining links in traceability recovery. Empir Softw Eng 22(3):996–1027
4. Pandian R, Kumar A (2019) Enhanced requirement traceability link using developer's updated activity. Modeling methods for business information systems analysis and design. IGI Global, London, pp 78–92
5. Zhao T, Cao Q, Sun, Q (2017) An improved approach to traceability recovery based on word embeddings. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE
6. Sale VM et al (2021) An effective approach for accuracy of requirement traceability in DevOps. Techno-societal 2020. Springer, Berlin, pp 623–637
7. Kuang H et al (2017) Analyzing closeness of code dependencies for improving IR-based Traceability Recovery. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE
8. Ali N et al (2019) Exploiting parts-of-speech for effective automated requirements traceability. Inf Softw Technol 106:126–141
9. Santos LRJ et al (2020) Improving traceability recovery between bug reports and manual test cases. In: Proceedings of the 34th Brazilian Symposium on Software Engineering
10. Chhabra JK (2017) Requirements traceability through information retrieval using dynamic integration of structural and co-change coupling. International conference on advanced informatics for computing research. Springer, Berlin
11. Saputri TRD, Lee S-W (2016) Ensuring traceability in modeling requirement using ontology based approach. In: Ch M (ed) Asia Pacific requirements engineering conference. Springer, Berlin
12. Kuang H et al (2019) Using frugal user feedback with closeness analysis on code to improve IR-based traceability recovery. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), IEEE
13. Gadelha G, Ramalho F, Massoni T (2021) Traceability recovery between bug reports and test cases-a Mozilla Firefox case study. Autom Softw Eng 28(2):1–46
14. Gazzawe F (2021) Requirement artefacts: finding the missing link, framework to support requirement traceability for software developers. Loughborough University, Loughborough
15. Niu N, Wang W, Gupta A (2016) Gray links in the use of requirements traceability. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering
16. Blasco D, Cetina C, Pastor O (2020) A fine-grained requirement traceability evolutionary algorithm: Kromaia, a commercial video game case study. Inf Softw Technol 119:106235
17. Ali N, Guéhéneuc YG, Antoniol G (2013) Trustrace: mining software repositories to improve the accuracy of requirement traceability links. IEEE Trans Softw Eng 39(5):725–741
18. Li T et al (2020) Combining machine learning and logical reasoning to improve requirements traceability recovery. Appl Sci 10(20):7253
19. Panichella A et al (2013) When and how using structural information to improve ir-based traceability recovery. In: 2013 17th European Conference on Software Maintenance and Reengineering, IEEE
20. Wang H et al (2021) Analyzing close relations between target artifacts for improving IR-based requirement traceability recovery. Front Inf Technol Electron Eng 22(7):957–968
21. Rodriguez DV, Carver DL (2019) Comparison of information retrieval techniques for traceability link recovery. In: 2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT), IEEE
22. Ogheneovo E, Japheth R (2016) Application of vector space model to query ranking and information retrieval. Int J Adv Res Comput Sci Softw Eng 6(5):42–47
23. Moran K et al (2020) Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp 873–885
24. Hayes J et al (2007) Requirements tracing on target (RETRO): improving software maintenance through traceability recovery. Innov Syst Softw Eng 3(3):193–202
25. Hayes JH, Dekhtyar A, Payne J (2018) The requirements tracing on target (RETRO). NET Dataset. In: 2018 IEEE 26th International Requirements Engineering Conference (RE), IEEE
26. Nattoch DJ et al (2005) A linguistic-engineering approach to large-scale requirements management. IEEE Softw 22(1):32–39
27. Jun L et al (2006) Poirot: a distributed tool supporting enterprise-wide automated traceability. In: 14th IEEE International Conference Requirements Engineering (RE'06) 2006
28. Lormans M, van Deursen A (2006) Can LSI help reconstructing requirements traceability in design and test? In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06) 2006
29. De Lucia A et al (2005) ADAMS Re-trace: a traceability recovery tool. In: 9th European Conference onSoftware Maintenance and Reengineering (CSMR'05) 2005
30. Florez JM (2019) Automated fine-grained requirements-to-code traceability link recovery. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE
31. Zhang Y, Wan C, Jin B (2016) An empirical study on recovering requirement-to-code links. In: 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE
32. McMillan C, Poshyvanyk D, Revelle M (2009) Combining textual and structural analysis of software artifacts for traceability link recovery. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, IEEE
33. Sundaram SK et al (2010) Assessing traceability of software engineering artifacts. Requir Eng 15(3):313–335
34. Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: Trends, techniques and applications. ACM Comput Surv 45(1):1–61
35. Mahmood S, Ahmed M, Alshayeb M (2013) Reuse environments for software artifacts: Analysis framework. In: 2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS), IEEE
36. Krueger CW (1992) Software reuse. ACM Comput Surv (CSUR) 24(2):131–183
37. Frakes WB, Kang K (2005) Software reuse research: status and future. IEEE Trans Softw Eng 31(7):529–536
38. Morisio M, Ezran M, Tully C (2002) Success and failure factors in software reuse. IEEE Trans Softw Eng 28(4):340–357

39. Keswani R, Joshi S, Jatain A (2014) Software reuse in practice. In: 2014 Fourth International Conference on Advanced Computing and Communication Technologies, IEEE
40. Sherif K, Appan R, Lin Z (2006) Resources and incentives for the adoption of systematic software reuse. Int J Inf Manag 26(1):70–80
41. Sommerville I (2011). In: Horton M (ed) Software engineering, 9th edn. Pearson Education, Inc., Addison-Wesley, ISBN: 0-13-703515-2, 978-0-13-703515-1
42. Heinemann L (2012) Effective and efficient reuse with software libraries. Technische Universität München, München
43. Mojica IJ et al (2013) A large-scale empirical study on software reuse in mobile apps. IEEE Softw 31(2):78–86
44. Buccella A et al (2013) Towards systematic software reuse of gis: insights from a case study. Comput Geosci 54:9–20
45. Lliteras AB et al (2020) Development, reuse, and repurposing of software artifacts in Digital Citizen Science. Are we reinventing the wheel? In: The VI Iberoamerican Conference of Computer Human Interaction, K.R.-P.a.K.O. Villalba-Condori, Editor. 2020: Arequipa, Perú, September 16–18, 2020
46. Capilla R et al (2019) Opportunities for software reuse in an uncertain world: From past to emerging trends. J Softw 31(8):217
47. Mikkonen T, Taivalsaari A (2019) Software reuse in the era of opportunistic design. IEEE Softw 36(3):105–111
48. Mäkitalo N et al (2020) On opportunistic software reuse. Computing 102(11):2385–2408
49. Krüger J, Berger T (2020) An empirical analysis of the costs of clone-and platform-oriented software reuse. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering
50. Zhang H et al (2016) Bing developer assistant: improving developer productivity by recommending sample code. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering
51. Wang Y, Feng Y, Martins R, Kaushik A, Dillig I, Reiss SP (2016) Type-directed code reuse using integer linear programming. arXiv preprint https://arxiv.org/abs/1608.07745. Accessed 26 Nov 2021
52. Colvin E, Kraft DH (2016) Fuzzy retrieval for software reuse. J Am Soc Inf Sci 67(10):2454–2463
53. Selby RW (2005) Enabling reuse-based software development of large-scale systems. IEEE Trans Software Eng 31(6):495–510
54. Spanoudakis G, Zisman A (2005) Software traceability: a roadmap. Handbook of software engineering and knowledge engineering: vol 3: recent advances. World Scientific, London, pp 395–428
55. Pohl K (1996) Process-centered requirements engineering. John Wiley and Sons Inc, New Jersey
56. Pohl K, Rupp C (2015). In: Barabas M (ed) Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam—foundation level—IREB compliant, 2nd edn. Rocky Nook Inc., ISBN: 978-1-937538-77-4
57. Egyed A, Grunbacher P (2002) Automating requirements traceability: beyond the record & replay paradigm. In: Proceedings 17th IEEE International Conference on Automated Software Engineering, IEEE.
58. Gotel OCZ, Finkelstein ACW (1994) An analysis of the requirements traceability problem. In: Proceedings of the First International Conference on Requirements Engineering (RE94)
59. Kaindl H (1993) The missing link in requirements engineering. ACM SIGSOFT Softw Eng Notes 18(2):30–39
60. Lindvall M, Sandahl K (1996) Practical implications of traceability. Software 26(10):1161–1180
61. Ramesh B, Dhar V (1992) Supportin systems development using knowledge captured during requirements engineering. IEEE Trans Softw Eng 9(2):498–510
62. Spanoudakis G et al (2004) Rule-Based Generation of Requirements Traceability Relations. J Syst Softw 72:105–127
63. Spanoudakis G et al (2004) Rule-based generation of requirements traceability relations. J Syst Softw 72(2):105–127
64. Elamin R, Osman R (2018) Implementing traceability repositories as graph databases for software quality improvement. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE
65. Antoniol G et al (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983
66. Nagano S, Ichikawa Y, Kobayashi T (2012) Recovering traceability links between code and documentation for enterprise project artifacts. In: 2012 IEEE 36th Annual Computer Software and Applications Conference, IEEE
67. Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: 25th International Conference on Software Engineering, Proceedings, IEEE
68. Mäder P, Gotel O, Philippow I (2009) Enabling automated traceability maintenance through the upkeep of traceability relations. In: Ch M (ed) European conference on model driven architecture-foundations and applications. Springer, Berlin
69. Mäder P, Gotel O (2012) Towards automated traceability maintenance. J Syst Softw 85(10):2205–2227
70. Cleland-Huang J, Chang CK, Ge Y (2002) Supporting event based traceability through high-level recognition of change events. In: Proceedings 26th Annual International Computer Software and Applications, IEEE
71. Pinheiro FA, Goguen JA (1996) An object-oriented tool for tracing requirements. IEEE Softw 13(2):52–64
72. Drivalos-Matragkas N et al (2010) A state-based approach to traceability maintenance. In: Proceedings of the 6th ECMFA Traceability Workshop
73. Lago P, Muccini H, Van Vliet H (2009) A scoped approach to traceability management. J Syst Softw 82(1):168–182
74. Mader P, Gotel O, Philippow I (2009) Getting back to basics: Promoting the use of a traceability information model in practice. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, IEEE
75. Elamin R, Osman R (2017) Towards requirements reuse by implementing traceability in agile development. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), IEEE
76. Ramesh B (1998) Factors influencing requirements traceability practice. Commun ACM 41(12):37–44
77. Ramesh B, Jarke M (2001) Toward reference models for requirements traceability. IEEE Trans Software Eng 27(1):58–93
78. Schwarz H, Ebert J, Winter A (2010) Graph-based traceability: a comprehensive approach. Softw Syst Model 9(4):473–492
79. Aizenbud-Reshef N et al (2006) Model traceability. IBM Syst J 45(3):515–526
80. Asuncion HU, François F, Taylor RN (2007) An end-to-end industrial software traceability tool. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
81. Witte R, Zhang Y, Rilling J (2007) Empowering software maintainers with semantic web technologies. European semantic web conference. Springer, Berlin
82. Sommerville I, Sawyer P (1997) RE: a good practice guide. John Wiley and Sons, New Jersey
83. Winkler S, von Pilgrim J (2010) A survey of traceability in requirements engineering and model-driven development. Softw Syst Model 9(4):529–565
84. Dorfman M, Thayer RH (eds) (1990) Standards, guidelines and examples on system and software requirements engineering. In:

IEEE computer society press tutorial volume 303; volume 305 of IEEE catalog. EHO. IEEE Computer Society Press. ISBN: 081865922X, 9780818659225
85. Bayer J, Widen T (2001) Introducing traceability to product lines. International workshop on software product-family engineering. Springer, Berlin
86. Dick J (2002) Rich traceability. IEEE Softw 22:14–16
87. Ali N, Gueheneuc YG, Antoniol G (2011) Requirements traceability for object oriented systems by partitioning source code. In: 2011 18th Working Conference on Reverse Engineering, IEEE
88. Tsuchiya R et al (2013) Recovering traceability links between requirements and source code in the same series of software products. In: Proceedings of the 17th International Software Product Line Conference
89. Ali N, Guéhéneuc YG, Antoniol G (2011) Trust-based requirements traceability. In: 2011 IEEE 19th International Conference on Program Comprehension, IEEE
90. Li Z et al (2015) Recovering traceability links in requirements documents. In: Proceedings of the Nineteenth Conference on Computational Natural Language Learning
91. Aung TWW, Huo H, Sui Y (2020) A literature review of automatic traceability links recovery for software change impact analysis. In: Proceedings of the 28th International Conference on Program Comprehension
92. Erata F et al (2017) Tarski: a platform for automated analysis of dynamically configurable traceability semantics. In: Proceedings of the Symposium on Applied Computing 2017.
93. Goknil A, Kurtev I, Van Den Berg K (2014) Generation and validation of traces between requirements and architecture based on formal trace semantics. J Syst Softw 88:112–137
94. Cleland-Huang J et al (2007) Best practices for automated traceability. Computer 40(6):27–35
95. Lormans M, Van Deursen A (2005) Reconstructing requirements coverage views from design and test using traceability recovery via LSI. In: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering
96. Harry B (2011) The importance of feedback in software development. https://devblogs.microsoft.com/bharry/the-importance-of-feedback-in-software-development/. Accessed 26 Nov 2021
97. Karahasanoviæ A et al (2005) Collecting feedback during software engineering experiments. Empir Softw Eng 10(2):113–147
98. Morales-Ramirez I, Perini A, Guizzardi RS (2015) An ontology of online user feedback in software engineering. Appl Ontol 10(3–4):297–330
99. Vargas EL et al (2018) Enabling real-time feedback in software engineering. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results
100. Johanssen JO et al (2019) How do practitioners capture and utilize user feedback during continuous software engineering? In: 2019 IEEE 27th International Requirements Engineering Conference (RE), IEEE
101. Goldberg ED (1989) Genetic algorithms in search optimization and machine learning. Addison-Wesley Longman Publishing Co., Inc, Boston, p 372
102. Mitchell M (1998) An introduction to genetic algorithms. MIT Press, Cambridge, p 209
103. Harbich S (2007) Einführung genetischer algorithmen mit anwendungsbeispiel. Universität Magdeburg, December, 2007
104. Araújo AA et al (2017) An architecture based on interactive optimization and machine learning applied to the next release problem. Autom Softw Eng 24(3):623–671
105. Kaliakatsos-Papakostas MA, Floros A, Vrahatis MN (2016) Interactive music composition driven by feature evolution. Springerplus 5(1):1–38
106. Khajeh M, Payvandy P, Derakhshan SJ (2016) Fashion set design with an emphasis on fabric composition using the interactive genetic algorithm. Fash Text 3(1):1–16
107. Johnston VS, Caldwell C (1997) Tracking a criminal suspect through face space with a genetic algorithm (Chapter: G8.3). In: Bäck T, Fogel DB, Michalewicz Z (eds) Handbook of Evolutionary Computation, release 97/1. IOP Publishing Ltd and Oxford University Press, pp G8.3:1–G8.3:8. https://b-ok.asia/book/703833/4a78e6
108. Kim H-S, Cho S-B (2000) Application of interactive genetic algorithm to fashion design. Eng Appl Artif Intell 13(6):635–644
109. Tokui N, Iba H (2000) Music composition with interactive evolutionary computation. In: Proceedings of the Third International Conference on Generative Art
110. Takagi H, Ohsaki M (2007) Interactive evolutionary computation-based hearing aid fitting. Evolut Comput IEEE Trans 11(3):414–427
111. Takagi H (2001) Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. Proc IEEE 89(9):1275–1296
112. Mwangi W, Cheruiyot W (2017) A survey of information retrieval techniques. Adv Netw 5(2):40
113. Saleem M, Minhas NM (2018) Information retrieval based requirement traceability recovery approaches-a systematic literature review. Univ Sindh J Inf Commun Technol 2(4):180–188
114. Gudivada VN, Rao D, Gudivada AR (2018) Information retrieval: concepts, models, and systems. Elsevier, Amsterdam
115. Abu-Salih B (2018) Applying vector space model (VSM) techniques in information retrieval for arabic language. arXiv preprint https://arxiv.org/abs/1801.03627. Accessed 26 Nov 2021
116. Van Nguyen T et al (2017) Combining word2vec with revised vector space model for better code retrieval. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE.
117. Marcus A, Maletic JI, Sergeyev A (2005) Recovery of traceability links between software documentation and source code. Int J Softw Eng Knowl Eng 15(05):811–836
118. Lapeña R et al (2018) Exploring new directions in traceability link recovery in models: The process models case. International conference on advanced information systems engineering. Springer, Berlin
119. Wang X, Lai G, Liu C (2009) Recovering relationships between documentation and source code based on the characteristics of software engineering. Electron Notes Theor Comput Sci 243:121–137
120. Tsuchiya R et al (2015) Recovering traceability links between requirements and source code using the configuration management log. IEICE Trans Inf Syst 98(4):852–862
121. De Lucia A et al (2013) Applying a smoothing filter to improve ir-based traceability recovery processes: An empirical investigation. Inf Softw Technol 55(4):741–754
122. Winkler S (2009) Trace retrieval for evolving artifacts. In: 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, IEEE
123. Panichella A, De Lucia A, Zaidman A (2015) Adaptive user feedback for ir-based traceability recovery. In: 2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability, IEEE
124. Alvarez-Rodríguez JM et al (2020) Semantic recovery of traceability links between system artifacts. Int J Softw Eng Knowl Eng 30(10):1415–1442
125. Rodriguez DV, Carver DL (2020) An IR-based artificial bee colony approach for traceability link recovery. In: 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI), IEEE

126. Tinnes C et al (2019) Ideas on improving software artifact reuse via traceability and self-awareness. In: 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST), IEEE
127. Mills C et al (2017) Predicting query quality for applications of text retrieval to software engineering tasks. ACM Trans Softw Eng Methodol (TOSEM) 26(1):1–45
128. Bäck T (1996) Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, Oxford, p 314
129. Chang-Yong L (2003) Entropy-Boltzmann selection in the genetic algorithms. IEEE Trans Syst Man Cybern Part B 33(1):138–149
130. Miller BL, Goldberg DE (1996) Genetic algorithms, selection schemes, and the varying effects of noise. Evol Comput 4(2):113–131
131. Baker JE (1985) Adaptive selection methods for genetic algorithms. Proceedings of the 1st international conference on genetic algorithms. L Erlbaum Associates Inc, Berlin, pp 101–111
132. Durillo JJ et al (2009) On the effect of the steady-state selection scheme in multi-objective genetic algorithms. In: Ehrgott M (ed) Evolutionary multi-criterion optimization: 5th international conference, EMO 2009, Nantes, France, April 7–10. Springer, Berlin, pp 183–197
133. Said MABM (2007) Comparison for selection techniques in genetic algorithm in faculty of computer systems and software engineering. University College of Engineering and Technology, Malaysia, p 180
134. Mayilvaganan M, Geethamani GS (2015) Analysis of roulette wheel selection and steady state selection using genetic algorithm techniques. Int J Comput Organ Trends 20(1):21–25
135. Jain A et al (2017) Information retrieval using cosine and jaccard similarity measures in vector space model. Int J Comput Appl 164(6):28–30
136. Havrlant L, Kreinovich V (2017) A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). Int J Gen Syst 46(1):27–36
137. Wohlin C et al (2000) Experimentation in software engineering-an introduction. Kluwer Academic Publishers, Doedrecht
138. Oliveto R et al (2010) On the equivalence of information retrieval methods for automated traceability link recovery. In: 2010 IEEE 18th International Conference on Program Comprehension, IEEE
139. Ben CE et al (2011) Towards a benchmark for traceability. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution
140. Antoniol G et al (2000) Information retrieval models for recovering traceability links between code and documentation. In: Proceedings of the International Conference on Software Maintenance (ICSM'00), IEEE Computer Society, p 40
141. Gethers M et al (2011) On integrating orthogonal information retrieval methods to improve traceability recovery. In: 27th IEEE International Conference on Software Maintenance (ICSM'11)
142. Mehlhorn K, Näher S (1999) LEDA: A platform for combinatorial and geometric computing. Cambridge University Press, Cambridge
143. Arora M, Kanjilal U, Varshney D (2016) Evaluation of information retrieval: precision and recall. Int J Indian Cult Bus Manag 12(2):224–236

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.