Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

On the Influence of Architectural Languages on Requirements Traceability

Manuel Ballarín*[1] | Lorena Arcega[1] | Vicente Pelechano[2] | Carlos Cetina[1]

[1]SVIT Research Group, Universidad San Jorge, Zaragoza, Spain
[2]Centro de Investigación en Métodos de Producción de Software (ProS), Universitat Politécnica de Valéncia, Valencia, Spain

Correspondence
*Manuel Ballarín, Email: mballarin@usj.es

Summary

Today, a considerable number of Architectural Languages (ALs) have been proposed for specifying and analyzing the architecture of software systems. Despite the popularity of different ALs, how ALs influence software system maintainability has not received much attention. One of the most important tasks in software maintenance is requirements traceability. Requirements traceability establishes links between requirements and other software artifacts, facilitating system maintenance. In this paper, we analyze the influence of ALs on requirements traceability. Taking into account the ALs used by the industry, we analyze how ALs influence traceability among requirements and architecture models. We conducted an evaluation with our industrial partner CAF. The results show significant differences in AL performance. We also analyze the results in terms of AL concepts, requirements model elements, and AL type in order to understand the performance differences. General-Purpose/Research Languages achieve the best results for all of the performance indicators, providing a mean precision value of 0.51, a recall value of 0.38, a combined F-measure of 0.40, and an MCC value of 0.33. Those ALs that influence engineers to use more generic and domain-independent terms to specify their architectures obtain the best results during requirements traceability. Our results have the potential to help AL designers to improve their languages and also to help practitioners make a more informed decision about whether or not a given AL meets their traceability needs.

KEYWORDS:
Architectural Languages, Software Maintenance, Requirement Traceability, Architecture Description Language

1 | INTRODUCTION

An Architectural Language (AL) [1] is a way to describe software systems[1]. ALs provide practitioners with a set of rules and common practices that help promote mutual communication, the embodiment of early design decisions, and the creation of a transferable abstraction of a system. Components and connectors are the main elements of ALs, they include rules and guidelines for well-formed architectures. AL's suitability varies for modeling particular kinds of systems (e.g., highly concurrent systems)

[1]Hereafter, we use the term architectural language, or AL, to refer to any form of expression used for architecture description. We use the term AL for the sake of clarity since, in the last few decades, several different definitions of the Architecture Description Language term have been proposed .

or particular aspects of a given system (e.g., its static properties)[2]. Furthermore, the set of aspects that are important enough to model varies from domain to domain[3].

A major driver for AL selection is system-specific: software development teams or individual software engineers select the AL that best fits the system to be realized, the domain of the system, or the specific project needs. Furthermore, other factors influence the selection of the AL for specifying a software system, such as the existence of a community that provides AL-related support, the skills and competences of current architects/developers, and tool support and costs[4].

In industrial scenarios, the usage of several ALs to specify different software systems is common. As observed in[5] and confirmed in[4], one of the reasons for the accumulation of so many ALs is the need to satisfy different stakeholders' concerns: "A language has to adequately capture design decisions judged fundamental by the system's stakeholders". Nevertheless, despite the popularity of different ALs, the question of how the usage of ALs influences software system maintainability has not yet received much attention.

One of the most important performed activities during the software system maintenance phase is requirements traceability[6]. Requirements traceability is concerned with the ability to relate requirements with other software artifacts (e.g., architecture models, source code) and establish the links between them during software development. Requirements traceability has been a subject of investigation for many years within the software engineering community[7,8]. Actively supporting traceability in a software development project can help to ensure the qualities of the software, such as maintainability. Being able to identify the links among requirements and architecture models is very critical in order to verify and trace non-reliable parts[9] and to decrease the expected defect rate in development software[10]. These traces can help software engineers better understand the system during software maintenance.

In this work, we analyze the influence of ALs on requirements traceability. Taking into account a diverse set of the ALs used by industry[4], we analyze how the use of these ALs influences traceability among requirements and architecture models. To do this, we rely on a Requirements Traceability to Architectural Language (RTAL) approach that is based on Latent Semantic Indexing (LSI)[11], which is the technique that has shown the best results for requirements traceability[12,13]. We evaluate the RTAL approach in a real-world industrial case study in the railway domain with our industrial partner CAF, a worldwide leader in railway manufacturing. CAF makes use of different ALs in order to describe software architecture for train control and management. We compared the effectiveness of RTAL for each of the ALs using the standard measurements accepted by the software engineering community: precision, recall, F-Measure, and Matthews Correlation Coefficient (MCC)[14,15]. Finally, we perform a statistical analysis of the results to provide quantitative evidence of the impact of the use of different ALs and to show that this impact is significant.

The results show a significant difference in AL performance. Y_DON is the AL that achieves the best results during requirements traceability, while EAST-ADL is the AL that achieves the worst results. We analyzed the results to understand what the best performing ALs bring to the table in order to achieve the best results. We analyzed the results in terms of AL concepts, requirements model elements, and AL type. To do this, we consider the classification presented by Taylor et al.[16], which distinguishes four AL categories based on the engineers' concerns and purposes. These categories are 1) General-Purpose/Research Languages (languages proposed to ease and improve the quality of software architectures); 2) Early Architecture Description Languages (languages proposed to provide interoperability, heterogeneity, and support composition and reusability); 3) Domain-and Style-specific Languages (languages proposed to support a particular set of tasks, as they are performed in a specific domain); and 4) Extensible architecture Description Languages (languages proposed to address currently languages deficiencies by providing a rich, extensible and flexible syntax for describing component interface types and the use of patterns and meta-information). Our analysis shows that General-Purpose/Research Languages (in this study: UML, Y_DON, SDL, and ARCHIMATE) achieved the best results for all of the performance indicators, providing a mean precision value of 0.51, a recall value of 0.38, a combined F-measure of 0.40, and an MCC value of 0.33. Those ALs that influence engineers to use more generic and domain-independent terms to specify their architectures obtained the best results during requirements traceability.

To the best of our knowledge, this paper presents the first investigation comparing ALs regarding their requirements traceability performance. Our paper contributes to understanding the influence of ALs on requirements traceability performance. Specifically, we claim that:

- There are significant performance differences among the widespread ALs. This is relevant for the software engineering community because requirement traceability is an essential task for software maintenance and evolution. Our results can help practitioners to choose the AL that best fits their needs in terms of requirement traceability.

- Our analysis of the results helps to understand the source of the performance differences among ALs. This has the potential to help AL designers to improve their ALs with regard to requirement traceability.

The remainder of the paper is structured as follows. Section 2 provides the background on architectural languages and requirements traceability in our industrial partner. Section 3 presents the RTAL approach in detail. Section 4 presents the evaluation. Section 5 provides insight into the discussion of the results. Section 6 presents the threats to validity. Section 7 presents the related work. Section 8 concludes the paper.

2 | BACKGROUND

Many Architectural Languages (ALs) can be found today. ALs must appropriately capture the design decisions that are considered to be essential by the system's stakeholders. An extensive review of the use of ALs is presented in[4]. The goal of its authors is to better understand the perceived strengths, limitations, and needs of practitioners regarding the use of ALs for software architecture modeling in the industry.

To analyze the influence of ALs during requirements traceability, we performed a study on the architectural languages used by previous researchers. First, we selected the ALs used by industry from a previous survey that analyzes different ALs[4]. The ALs are listed in Table 1, these ALs were most commonly used by 48 engineers from 40 different IT companies. Initially, we examined the 23 architectural languages that are considered to be the most widely used in the industry (as is stated in[4]). However, we had to discard some of them for different reasons: some of them were not possible to find (even by sending emails to the authors of the language), others do not have tool support, and others did not have follow-up and therefore are currently outdated. Finally, we got a list with eleven ALs: AADL[37], Acme[38], ARCHIMATE[39], EAST-ADL[40], MIND[27], Modelica[28], PCM[30], SDL[41], UML[35], xADL[42], and Y_DON[43].

Table 1 shows a list of the ALs considered for this study (highlighted in grey). This list includes the tool used, a link for more information, and the popularity of each AL. The popularity of an AL is based on how many engineers from the total of 48 engineers interviewed used that AL (see the study conducted by Malavolta et al.[4]).

The running example and the evaluation in this paper are performed using the products of our industrial partner, CAF. CAF is a worldwide provider of railway solutions. Their trains can be seen all over the world and in different forms (regular trains, subway, light rail, monorail, etc.). A train unit is furnished with multiple pieces of equipment throughout its vehicles and cabins. These pieces of equipment are often designed and manufactured by different providers, and their aim is to carry out specific tasks for the train. Some examples of these devices are the traction equipment, the compressors that feed the brakes, the pantograph that harvests power from the overhead wires, or the circuit breaker that isolates or connects the electrical circuits of the train. The control software of the train unit is in charge of making all the equipment cooperate to achieve the train functionality while guaranteeing compliance with the specific regulations of each country.

Figure 1 depicts a simple example of three different architecture models that realize the same requirement and are specified through different ALs. For this example, we have chosen the three top-ranked ALs used by industry[4]: the top architecture model is specified through UML, the center architecture model is specified through ARCHIMATE, and the bottom architecture model is specified through AADL. UML model contains nine model elements, including five classes and four associations (connectors); ARCHIMATE model contains 13 model elements, including 5 application components, 4 application interfaces, and 4 *used by* connectors; and AADL model contains 17 model elements including five process elements, 8 ports, and 4 connectors. Some of the architecture languages allow the definition of the requirements to be closer to the domain. We consider these languages more abstract because they can be considered closer to reality. For example, the elements that appear in the ARCHIMATE language allow requirements to be defined in a way that is close to the domain, unlike the UML. Therefore, ARCHIMATE is more abstract than UML. ARCHIMATE uses application components, application interfaces, and *used by* connectors, while UML only uses classes and associations.

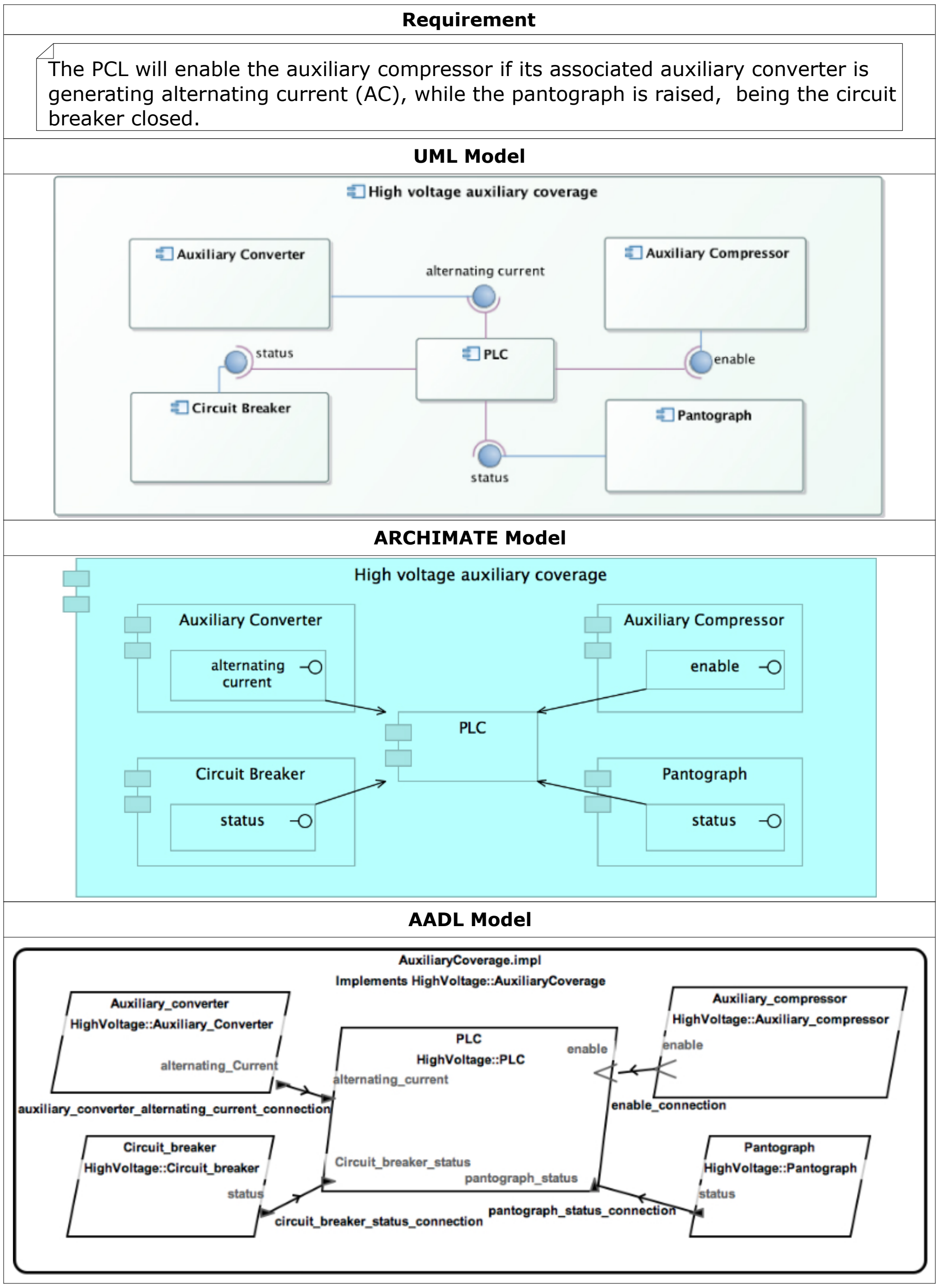
The requirement (Figure 1, top) describes the behavior of high voltage auxiliary coverage in the railway domain of the industrial partner. In natural language, this requirement is described as follows: *The PLC will enable the auxiliary compressor if its associated auxiliary converter is generating alternating current (AC), while the pantograph is raised, being the circuit breaker closed.*

TABLE 1 List of the Architectural Languages considered for this study

Architecture Language	Tool	Learn more at	Popularity
AADL	OSATE 2 Open Source AADL Tool Environment	The Open Source AADL Tool Environment (OSATE)[17]	7/48
ABACUS	ABACUS 7 (not tool support)	https://www.avolutionsoftware.com/abacus/	2/48
Acme	AcmeStudio	AcmeStudio: Supporting style-centered architecture development[18]	2/48
ArchiMate	Archi – Open Source ArchiMate Modelling	Archi-Open Source Archimate Modelling[19]	6/48
CCL	CCLi (not tool support)	Snapshot of CCL: A language for predictable assembly[20]	2/48
Darwin	Darwin Tool (Not available)	Specifying distributed software architectures[21]	2/48
EAST-ADL	MetaEdit+ 5.5	EAST-ADL: An architecture description language for automotive software-intensive systems[22]	3/48
HOOD	not found	Defining software architectures using the Hierarchical Object-Oriented Design method (HOOD)[23]	2/48
IAF	not found	The integrated architecture framework explained: why, what, how[24]	2/48
KISS	not found	Knowledge industry survival strategy (KISS)[25]	2/48
Koala	not found	The Koala Component Model for Consumer Electronics Software[26]	2/48
MIND	MindEd 0.2.1 Eclipse Plugin	The MIND project[27]	2/48
Modelica	OPENMODELICA	Modelica—A unified object-oriented language for system modeling and simulation[28]	2/48
OLAN	not found	The olan architecture definition language[29]	2/48
PCM	Palladio-Bench	The Palladio Component Model[30]	2/48
RAPIDE	not found	Specification and analysis of system architecture using Rapide[31]	4/48
SAMM	not found	Q-ImPrESS project deliverable D2. 1: service architecture metamodel (SAMM)[32]	2/48
SDL	PragmaDev Studio	Model-based testing: an approach with SDL/RTDS and DIVERSITY[33]	2/48
SDO	not found	https://www.osoa.org/display/Main/SDO+Resources	2/48
SLX	Wolverine Software Corporation (Not available)	Inside discrete-event simulation software: how it works and why it matters[34]	2/48
UML	UML Designer 9.0	The unified modeling language user guide[35]	38/48
xADL	ArchStudio 4	Archstudio 4: An architecture-based meta-modeling environment[36]	2/48
Y_DON	Visual Paradigm's DFD tool	https://www.visual-paradigm.com/	2/48

3 | REQUIREMENTS TRACEABILITY TO ARCHITECTURAL LANGUAGE (RTAL)

By targeting a real-world industrial case study, the goal of the presented work is to analyze the influence of Architectural Languages (ALs) in one of the most commonly performed activities during the software system maintenance phase: requirements traceability.


Requirement

The PCL will enable the auxiliary compressor if its associated auxiliary converter is generating alternating current (AC), while the pantograph is raised, being the circuit breaker closed.

FIGURE 1 Example of a requirement specified through the top-ranked Architectural Languages in Industry

In this work, we use an RTAL approach that is based on Information Retrieval (IR) techniques. IR techniques index the documents in a document space as well as the queries by extracting information about the occurrences of terms within them. This information is used to define similarity measures between queries and documents. In the case of RTAL, this similarity measure is used to identify that a traceability link might exist between two artifacts, one of which is used as a query[44]. Specifically, the RTAL approach used in this work relies on Latent Semantic Indexing (LSI)[11], which is the IR technique that achieves the best RTAL results[12,13] and that has been successfully applied to different kinds of software artifacts in different contexts[44,45,15].

LSI is an automatic mathematical/statistical technique that analyzes relationships between queries and documents. Specifically, given a certain requirement-model pair as input for LSI, the RTAL approach uses the outcome produced from the LSI to

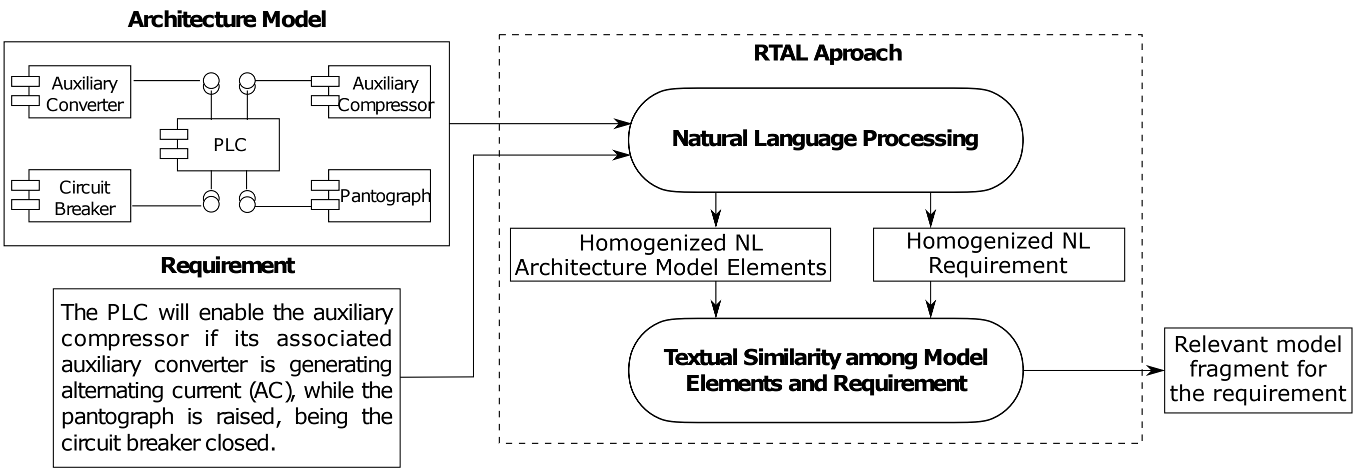

Architecture Model
Auxiliary Converter
Auxiliary Compressor
PLC
Circuit Breaker
Pantograph
Requirement
The PLC will enable the auxiliary compressor if its associated auxiliary converter is generating alternating current (AC), while the pantograph is raised, being the circuit breaker closed.
RTAL Aproach
Natural Language Processing
Homogenized NL Architecture Model Elements
Homogenized NL Requirement
Textual Similarity among Model Elements and Requirement
Relevant model fragment for the requirement

FIGURE 2 Overview of Requirements Traceability to Architectural Language (RTAL)

build a model fragment that serves as a candidate for realizing the requirement. The LSI technique is based on textual similarity in order to determine which model elements are closer to the provided requirement. In this way, the construction indexing of the artifacts (requirement and architecture model) is preceded by a text homogenization phase. During this phase, different Natural Language Processing (NLP) techniques are applied.

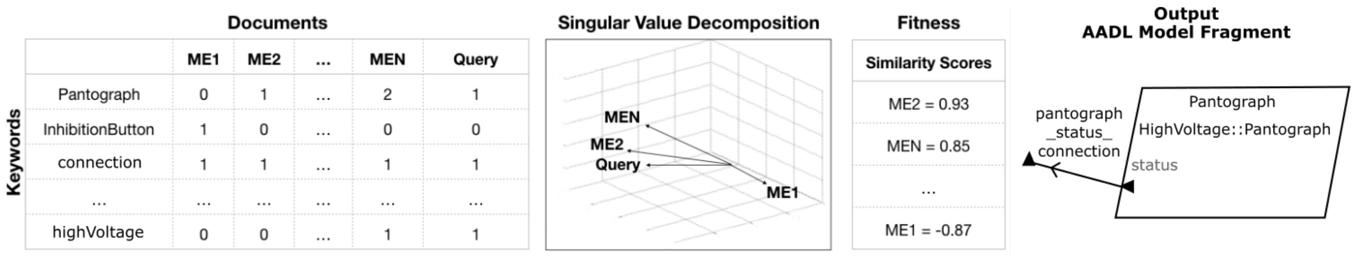
Figure 2 presents an overview of the RTAL approach. The left part shows the inputs for the approach: an architecture model specified in an architectural language and a requirement in natural language. The center shows a simplified representation of the main steps. The rounded rectangular boxes represent the different steps of the RTAL approach. The 'Natural Language Processing' step homogenizes the natural language from the model and the requirement. Finally, the 'Textual Similarity among Model Elements and the Requirement' step recover the traceability links between requirements and architecture models based on textual similarity. As output, the RTAL approach provides a model fragment relevant to the requirement. The following sections describe the RTAL approach in detail.

3.1 | Natural Language Processing

First, the RTAL approach deals with homogenizing the natural language from the requirement and the natural language from the elements that form the architecture model. Well-known Natural Language Processing (NLP) techniques are applied: the Parts-of-Speech tagging technique [7], and Lemmatizing techniques [23]. Thanks to these techniques, the language of both the requirements and the ALs is unified, avoiding verb tenses, noun plurals, and strange word forms that negatively interfere with the RTAL process.

The inclusion of domain experts, particularly software engineers, in traceability processes is a widely discussed topic within the SE community. It is often regarded as beneficial to have some sort of domain knowledge embedded in automated traceability systems, particularly in areas that are related to software reuse and software variability. Some of the techniques derived from humans interacting with traceability processes are Domain Term Extraction and Stopword Removal.

In order to carry out these techniques, RTAL Engineers provide two separate lists of terms: a list of terms (both single-word terms and multiple-word terms) that belong to the domain and that must always be kept for analysis and a list of irrelevant words that can appear throughout the entirety of the specification documents and that have no value whatsoever for the analysis. Both kinds of terms can be automatically filtered in or out of the final query, depending on the needs of the domain experts. For example, a list of relevant domain terms contains words such as *Pantograph*, *Multiplexer*, *Left door*, and/or *CCTV system*, among others. By contrast, a list of irrelevant words contains words such as *Trigger*, *State*, *Time*, and/or *Status*; among other words.



Documents

Keywords	ME1	ME2	...	MEN	Query
Pantograph	0	1	...	2	1
InhibitionButton	1	0	...	0	0
connection	1	1	...	1	1
...	...	...	...	...	...
highVoltage	0	0	...	1	1

Fitness

Similarity Scores
ME2 = 0.93
MEN = 0.85
...
ME1 = -0.87

FIGURE 3 Requirements Traceability to Architecture Languages (RTAL) through Latent Semantic Indexing Example

3.2 | Textual Similarity among Model Elements and the Requirement

After homogenizing the natural language from the requirement and the natural language from the elements that conform the architecture model, LSI recovers the traceability links between requirements and architecture models. LSI is an automatic mathematical/statistical technique that analyzes relationships between *queries* and *documents* (bodies of text). It constructs vector representations of both a user *query* and a corpus of text *documents* by encoding them as a *term-by-document co-occurrence matrix* and analyzes the relationships between those vectors to get a similarity ranking between the *query* and the *documents* (see Figure 3).

Figure 3 shows an example of a *term-by-document co-occurrence matrix*, with values associated with our case study, the vectors, and the resulting ranking. An overview of the elements of the matrix is provided below.

- Each row in the matrix (*term*) stands for each of the words that compose the processed requirement and natural language representation of the input architecture model. Figure 3 shows a set of representative words in the domain, such as 'Pantograph', as the *terms* of each row.
- Each column in the matrix (*document*) stands for one of the model elements extracted from the input architecture model. Figure 3 shows identifiers in the columns such as 'ME1' or 'ME2', which represent the *documents* of those specific architecture model elements.
- The final column stands for the *query*, which is one requirement.
- Each cell in the matrix contains the frequency with which the *term* of its row appears in the *document* denoted by its column. For instance, in Figure 3, the *term* 'Pantograph' appears twice in the 'MEN' *document* and once in the *query*.

Vector representations of the *documents* and the *query* columns are obtained by normalizing and compositing the *term-by-document co-occurrence matrix* using Singular Value Decomposition (SVD)[46,11]. SVD is a form of factor analysis, or more properly, the mathematical generalization of which factor analysis is a special case. In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values, another describes the original column entities in the same way, and the third is a diagonal matrix that contains scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed.

Figure 3 presents a three-dimensional graph of the SVD. The graph shows the vectorial representations of some of the matrix columns. For legibility reasons, only a small set of columns is presented. To measure the degree of similarity between vectors, the RTAL approach calculates the cosine between the *query* vector and the *document* vectors. Cosine values that are closer to 1 denote a higher degree of similarity, and cosine values that are closer to -1 denote a lower degree of similarity. Similarity increases as vectors point in the same general direction (as more *terms* are shared between *documents*). Through this measurement, the model elements are ordered according to their degree of similarity to the requirement.

The relevancy ranking (shown in Figure 3) is produced according to the calculated degrees of similarity. In this example, LSI retrieves 'ME2' and 'MEN' in the first and second position of the relevancy ranking since the *query-documents* cosines are '0.93' and '0.85', implying a high degree of similarity between the model elements and the requirement. In contrast, the 'M1' Model Element is returned in a lower position of the ranking because its *query-document* cosine is '-0.87', implying a low degree of similarity.

From the ranking of all the model elements, those that have a similarity measure greater than x must be taken into account. The heuristic that the RTAL approaches use, and that is used in other works[15,47] is $x = 0.7$. This value corresponds to a 45°

angle between the corresponding vectors. Nevertheless, the selection of this threshold is an issue that is still under study, and its proper parametrization has not yet been tackled in architecture models.

Following this principle, the model elements with a similarity measure equal or superior to $x = 0.7$ are taken to form a model fragment, which is a candidate for realizing the requirement. Through the example provided in Figure 3, ME2 and MEN are the model elements that are part of the model fragment obtained for the requirement, since their cosine values are superior to the threshold. The model elements below the threshold, except ME1, are not shown in the ranking for reasons of space and understandability. The model fragment generated in this manner is the final output of the RTAL approach. The right part of Figure 3 shows an example of a model fragment specified through AADL. The model fragment reference the complete AADL model that is shown in Figure 1. Words, such as 'Pantograph' or 'HighVoltage', can be seen in the visual representation, however, some others are part of the properties and do not have a visual representation.

4 | EVALUATION

In this section, we aim to clearly establish the scope of our work and to determine the key research questions that we must tackle and bear in mind when designing our experiment. The following research questions (RQ) arise from the described problem.

RQ$_1$: How do the different Architectural Languages influence traceability among requirements and architecture models?

RQ$_2$: Is the difference in performance between Architectural Languages significant?

RQ$_3$: How much is the quality of the solution influenced by each Architectural Language?

Answering RQ$_1$ allows us to compare the performance results (in terms of recall, precision, F-measure, and MCC) of each of the Architectural Languages in requirements traceability. Answering RQ$_2$ allows us to provide formal and quantitative evidence (using the Quade test and the Holm's post hoc analysis) to determine whether or not the difference in performance is significant. Answering RQ$_3$ allows us to assess (through effect size measure, Vargha and Delaney's $\hat{A}_{12}$) how much the quality of the solutions is influenced by each Architectural Language.

In the following subsections, we introduce the experimental setup and the case study used, and we provide details of the implementation of our evaluation. Finally, we present the results and the statistical analysis of our evaluation.

4.1 | Experiment Setup

Figure 4 shows an overview of the process that was followed to evaluate the approach. The left part shows the artifacts, which are provided by the industrial partner: requirements, architecture models, and approved traceability between requirements and architecture models. The set of requirements is specified through each of the Architectural Languages selected for this study (a set of those most relevant used by industry[4]).

A dedicated team of industrial experienced specialists was made available to us by the industrial partner to perform the modeling of different requirements using different ALs. The team consisted of five system architects: one is experienced in AADL, EAST-ADL, and Modelica, two are experienced in MIND, PCM, ACME, and xADL, one is experienced in Y_DON and ARCHIMATE, and one is experienced in SDL and UML. They were in charge of realizing a set of requirements using the 11 different ALs presented in this paper. After realizing the different models, the five system architects reviewed all of the models.

As a result, the documentation includes eleven different architecture models that specify the same set of requirements but using different ALs. As shown in Figure 4, the requirements and architecture model conform the input of the RTAL approach, and approved traceability plays the role of the oracle.

The RTAL approach takes those inputs and obtains a model fragment for each requirement. The generated model fragments are compared with the oracle. Once the comparisons are performed, a confusion matrix is calculated. A confusion matrix is a table that is often used to describe the performance of a classification model (in this case, RTAL) on a set of test data (the resulting model fragments) for which the true values are known (the oracle). In our case, each solution output by the RTAL approach is a model fragment that is composed of a subset of the model elements that are part of the architecture model. Since the granularity is at the level of model elements, the presence or absence of each model element is considered to be a classification. The confusion matrix will distinguish between two values (TRUE or presence and FALSE or absence). We obtain a confusion matrix for each requirement predicted by comparing the actual model fragment that corresponds to the requirement (obtained from the oracle and considered the ground truth) and the predicted model fragment for the requirement. The confusion matrix arranges the results of the comparison into four categories:

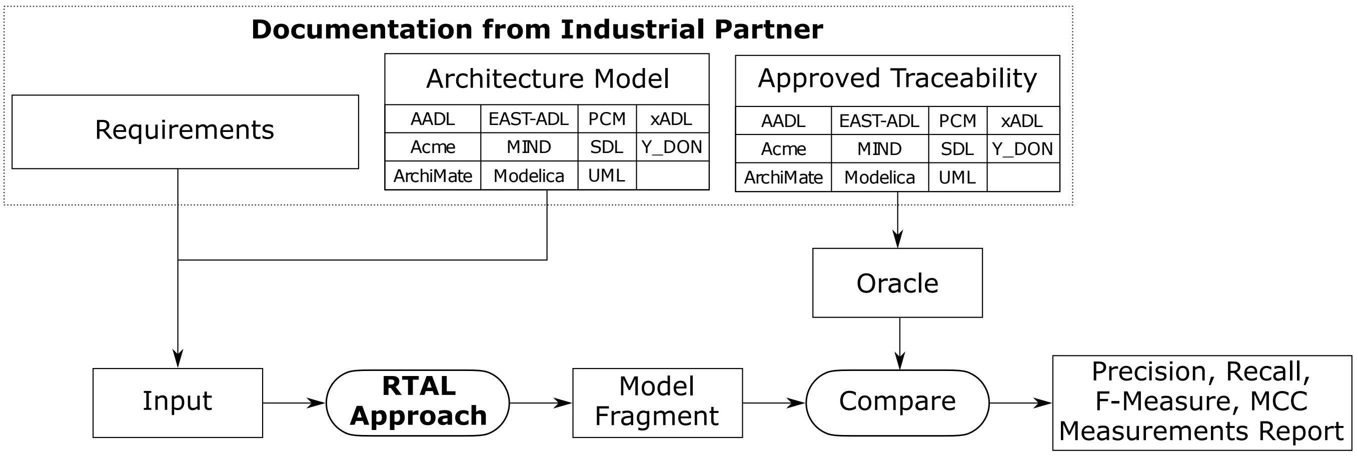


FIGURE 4 Experimental Setup

- **True Positive (TP)**: an element that is predicted as present (in the solution) and is present in the real scenario (the oracle).
- **False Positive (FP)**: an element that is predicted as present (in the solution) but is not present in the real scenario (the oracle).
- **True Negative (TN)**: an element that is predicted as not present (in the solution) and is not present in the real scenario (the oracle).
- **False Negative (FN)**: an element that is predicted as not present (in the solution) but is present in the real scenario (the oracle).

Then, some performance measurements are derived from the values in the confusion matrix. Specifically, a report that includes four performance measurements (Precision, Recall, F-Measure, and Matthews Correlation Coefficient) is created for the case study.

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

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

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

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

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

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2TP + FP + FN} \tag{3}$$

However, none of these previous measures correctly handle negative examples (TN). The MCC is a correlation coefficient between the observed and predicted binary classifications that takes into account all of the observed values (TP TN, FP, FN) and is defined as follows:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4}$$

Precision and recall values can range from 0 to 1. A precision value equal to 0 means that no single model element from the solution is the oracle while a precision value equal to 1 means that all of the model elements from the solution are present in the oracle. A recall value equal to 0 means that no single model element from the realization of the requirement obtained from the oracle is present in the model fragment of the solution while a recall value equal to 1 means that all of the model elements from the oracle are present in the solution. A precision and recall values equal to 1 implies that both the solution and the requirement from the oracle are the same. MCC values can range between -1 and 1. An MCC value equal to -1 means that there is no correlation between the prediction and the solution, an MCC value equal to 1 means that the prediction is perfect, and an MCC value equal to 0 means that the prediction is random.

4.2 | Case Study

For our evaluation, our industrial partner, CAF, provided us with natural language requirements and architecture models. The data that support the findings of this study are openly available on the SVIT Research Group web at https://svit.usj.es/al-tlr-data/.

The software of a train is specified through more than 500 natural language requirements, with an approximate average of 50 words per requirement. The architecture models are specified with an average of 330 total model elements. We followed the experimental setup shown in Figure 4. For this case study, we used a subset of 16 requirements (randomly selected). We executed an independent run for each of the 16 requirements for each of the 11 ALs considered for this study, i.e., 16 (requirements) x 11 (AL) = 176 independent runs.

4.3 | Implementation details

This approach was implemented within different environments, each of which depends on a specific software architecture. The IR techniques used to process the language were implemented using OpenNLP[48] for the POSTagger. LSI was implemented using the Efficient Java Matrix Library (EJML)[49].

In addition, in order to specify the requirements through different ALs, we used the following tools (also specified in Table 1): OSATE 2.2.3 to specify AADL models, AcmeStudio to specify Acme models, Archi 4.0.3 to specify ArchiMate models, MetaEdit+ 5.5 to specify EAST-ADL models, MindEd 0.2.1 Eclipse Plugin to specify MIND models, OpenModelica to specify Modelica models, Palladio-Bench to specify PCM models, PragmaDev Studio to specify SDL models, UML Designer 9.0 to specify UML models, ArchStudio 4 tool to specify xADL models, and Visual Paradigm's DFD tool to specify Y_DON models.

4.4 | Results

This subsection presents the results obtained once the RTAL approach was executed for each of the ALs selected for this study. Appendix A presents the charts with the precision and recall results for each requirement for our real-world case study and the eleven ALs. A dot in the graph represents the average result of precision (x-axis) and recall (y-axis) for each of the requirements in CAF.

RQ$_1$ answer. In Table 2, we outline the results that are aggregated for each AL in our case study. We also show the F-Measure and MCC performance indicators. Similarly, Figure 5 shows the box plots obtained from those results. The AL that achieved the best results is Y_DON, attaining 1 in precision, 0.65 in recall, 0.78 in F-measure, and 0.75 in MCC. The second-best result in precision was obtained by MIND, reaching 0.69; however, the recall value was very low, 0.04. The third-best result in precision was obtained by ARCHIMATE, reaching 0.53, which also obtained the second-best result in recall, 0.39. In contrast, both EAST-ADL and MODELICA obtained the worst results in all of the measurements. EAST-ADL obtained 0.06 in precision, 0.02 in recall, 0.02 in F-measure, and -0.49 in MCC; MODELICA obtained 0.01 in precision, 0.01 in recall, 0.01 in F-Measure, and -0.26 in MCC.

4.5 | Statistical Analysis

In order to properly compare the results obtained in RTAL with the different architectural languages, we performed the statistical analysis of the results following the criteria of[50]. We measured statistical significance and effect size. The statistical significance allows us to provide formal and quantitative evidence that architectural languages have an impact on the metrics used for the comparison. The effect size allows us to present that the differences in the results are significant in practice.

4.5.1 | Statistical Significance

A statistical test is run to assess whether there is enough empirical evidence to claim that there is a difference between the approaches (e.g., approach A is better than approach B). First, we need to define two hypotheses: the null hypothesis (H_0) and the alternative hypothesis (H_1). In contrast to the alternative hypothesis, the null hypothesis states that there is no difference between the approaches. Then, the statistical test verifies whether the null hypothesis could be rejected.

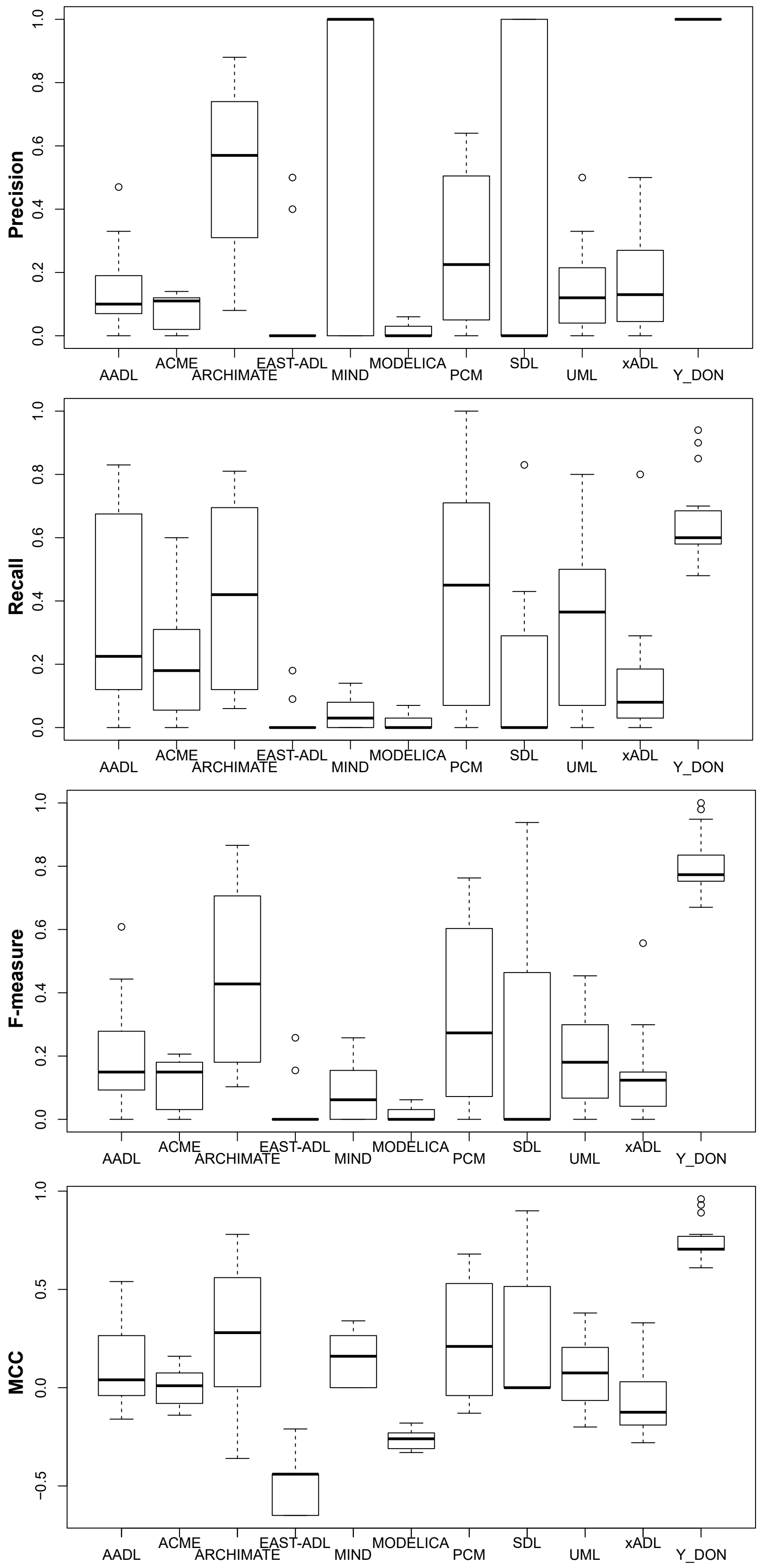


FIGURE 5 Box plots of the Precision, Recall, F-Measure, and MCC values obtained by each of the Architectural Languages

TABLE 2 Mean Values and Standard Deviations for Precision, Recall, F-Measure, and MCC for each Architectural Language.

	Precision $\pm\ \sigma$	**Recall** $\pm\ \sigma$	**F-measure** $\pm\ \sigma$	**MCC** $\pm\ \sigma$
AADL Models	0.13 ± 0.13	0.37 ± 0.32	0.19 ± 0.17	0.10 ± 0.21
ACME Models	0.08 ± 0.05	0.21 ± 0.19	0.11 ± 0.08	0.00 ± 0.10
ARCHIMATE Models	0.53 ± 0.25	0.39 ± 0.28	0.43 ± 0.27	0.28 ± 0.33
EAST-ADL Models	0.06 ± 0.15	0.02 ± 0.05	0.02 ± 0.07	-0.49 ± 0.13
MIND Models	0.69 ± 0.48	0.04 ± 0.05	0.08 ± 0.08	0.15 ± 0.12
MODELICA Models	0.01 ± 0.02	0.01 ± 0.02	0.01 ± 0.02	-0.26 ± 0.05
PCM Models	0.28 ± 0.24	0.42 ± 0.35	0.31 ± 0.26	0.24 ± 0.29
SDL Models	0.38 ± 0.50	0.15 ± 0.24	0.20 ± 0.29	0.22 ± 0.31
UML Models	0.15 ± 0.14	0.33 ± 0.25	0.19 ± 0.15	0.08 ± 0.18
xADL Models	0.17 ± 0.15	0.15 ± 0.20	0.13 ± 0.14	-0.06 ± 0.16
Y_DON Models	1.00 ± 0.00	0.65 ± 0.13	0.78 ± 0.09	0.75 ± 0.10

The rejection of the null hypothesis is performed taking into account the $p-value$ provided by the statistical test. This value can range between 0 and 1. Assuming that the null hypothesis is true, the closer the value is to 0, the lower the probability of obtaining results at least as extreme as the observed ones. For the research community, a $p-value$ below 0.05 is accepted to indicate that the null hypothesis can be considered false[50].

To select the test to follow, we took into account the distribution and the nature of our data. Our data does not follow a normal distribution in general. In addition, our data is extracted from a real environment. Since our data does not follow a normal distribution, our analysis requires the use of non-parametric techniques. There are several tests for analyzing this kind of data; however, the Quade test shows that it is more powerful than the others when working with real data, i.e., extracted from a real environment[51].

RQ$_2$ answer. The $p-values$ obtained in the test are $\ll 2.2x10^{-16}$ for precision and MCC, and $1.705x10^{-10}$ for recall; the statistics values obtained are 13.962, 8.1814, and 15.314 for precision, recall, and MCC, respectively. Since the $p-values$ are smaller than 0.05, we can reject the null hypothesis. Consequently, we can state that there are differences among the algorithms for the performance indicators of precision, recall, and MCC.

Next to the Quade test, we perform an additional post hoc analysis to test which algorithm gives the best performance. With the Holm's post hoc analysis, we compare each algorithm against all other alternatives to indicate whether exist significant differences between the results of a specific pair of algorithms.

Table 3 shows the $p-Values$ of Holm's post hoc analysis for the case study and the performance indicators for each pair of algorithms. The majority of the $p-Values$ obtained by Y_DON are smaller than their corresponding significance threshold value (0.05), indicating that the differences in performance between this AL and the rest of the ALs are significant. However, when we compare AADL, ACME, SDL, UML, or xADL with the rest of the ALs, the majority of the values are greater than the threshold. This indicates that the differences between those ALs could be due to mere chance and are not significant.

4.5.2 | Effect size

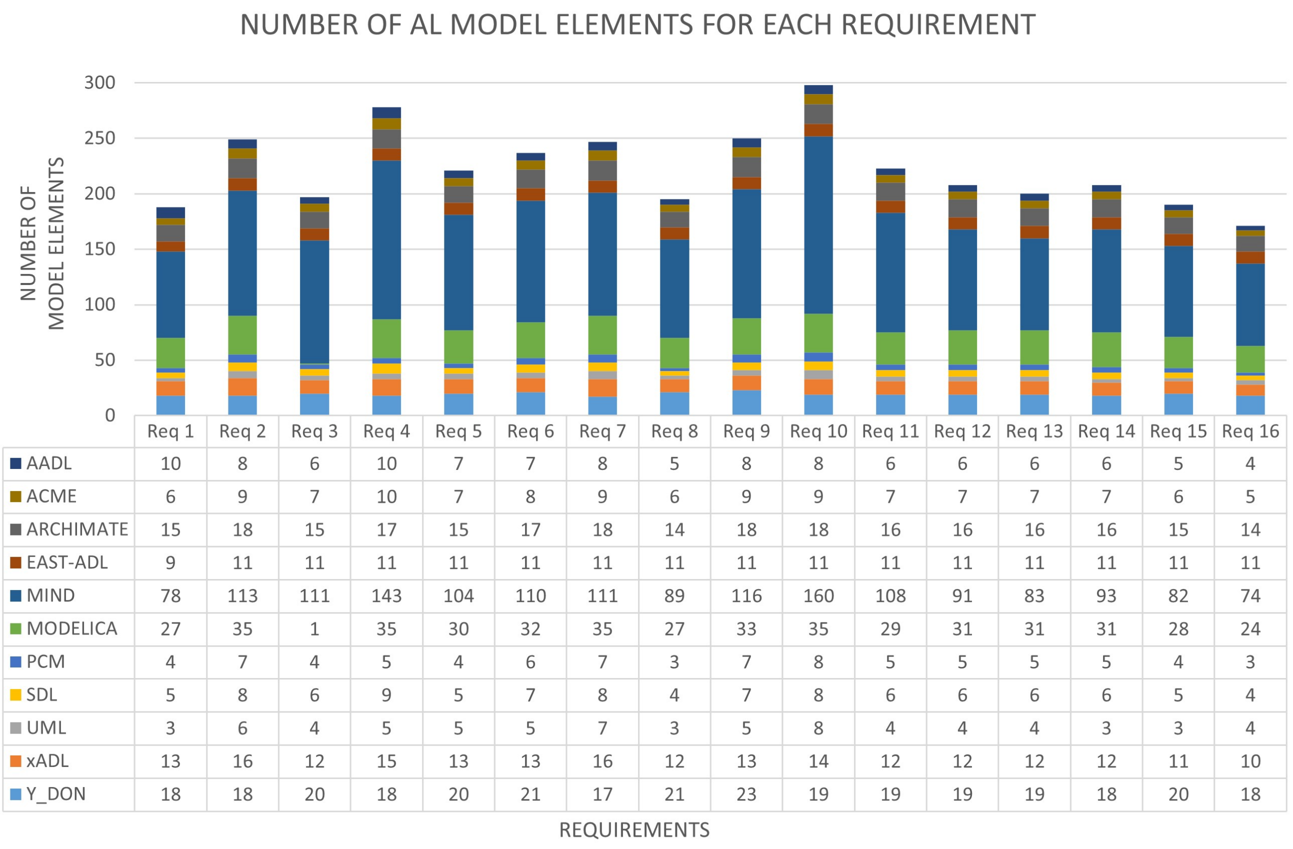
The effect size assesses if an algorithm is statistically better than another and the magnitude of the improvement. For a non-parametric effect size measure, we use Vargha and Delaney's $\hat{A}_{12}$[52,53]. $\hat{A}_{12}$ measures the probability that running one algorithm yields higher values than running another algorithm. A value of 0.5 means that two algorithms are equivalent.

For example, we want to measure the effect size of an algorithm A against an algorithm B. An $\hat{A}_{12}$ value equal to 0.7 means that the algorithm A would obtain better results in 70% of the runs. Similarly, an $\hat{A}_{12}$ value equal to 0.3 means that the algorithm B would obtain better results in 70% of the runs.

RQ$_3$ answer. Table 3 shows the values of the effect size statistics. In general, the largest differences were obtained between Y_DON and the rest of the ALs, where Y_DON achieves the best results every time. When comparing ARCHIMATE, MIND, and PCM with the rest of the ALs, the differences are not so large, but they obtain better results the majority of times. However, when we compare EAST-ADL with the rest of the ALs, EAST-ADL gets worse results every time.

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

	Holm's			$\hat{A}_{12}$		
	Precision	Recall	MCC	Precision	Recall	MCC
AADL vs. ACME	1	1	1	0.58203	0.62695	0.62695
AADL vs. ARCHIMATE	0.00574	1	1	0.08398	0.49609	0.31641
AADL vs. EAST-ADL	0.36681	0.00162	0.00015	0.79687	0.87891	1
AADL vs. MIND	0.08503	0.39920	1	0.28320	0.83203	0.36133
AADL vs. MODELICA	0.18268	0.00613	0.01623	0.87109	0.87109	1
AADL vs. PCM	1	1	1	0.33594	0.45898	0.35352
AADL vs. SDL	1	0.62333	1	0.56641	0.71484	0.32227
AADL vs. UML	1	1	1	0.42968	0.54102	0.52539
AADL vs. xADL	1	1	1	0.41602	0.70508	0.73047
AADL vs. Y_DON	0.98405	$\ll 2.2x10^{-16}$	$1.8x10^{-05}$	0	0.26953	0
ACME vs. ARCHIMATE	0.00057	0.81473	1	0.03516	0.30859	0.22852
ACME vs. EAST-ADL	1	0.15266	0.00011	0.76563	0.84375	1
ACME vs. MIND	0.01377	1	1	0.27344	0.77344	0.18359
ACME vs. MODELICA	0.67573	0.38498	0.01322	0.81641	0.82813	1
ACME vs. PCM	1	0.83358	1	0.27344	0.33008	0.25781
ACME vs. SDL	1	1	1	0.54688	0.63477	0.33203
ACME vs. UML	1	1	1	0.33398	0.36328	0.36914
ACME vs. xADL	1	1	1	0.34375	0.63281	0.68555
ACME vs. Y_DON	$1.9x10^{-08}$	0.03101	$2.5x10^{-05}$	0	0.04297	0
ARCHIMATE vs. EAST-ADL	$1.2x10^{-07}$	$5.0x10^{-05}$	$2.8x10^{-08}$	0.94531	0.96875	0.99219
ARCHIMATE vs. MIND	1	0.03806	1	0.31250	0.94531	0.58984
ARCHIMATE vs. MODELICA	$2.7x10^{-08}$	0.00022	$1.1x10^{-05}$	1	0.99219	0.93750
ARCHIMATE vs. PCM	0.10538	1	1	0.77539	0.48438	0.54688
ARCHIMATE vs. SDL	0.00739	0.06649	1	0.62500	0.79883	0.56445
ARCHIMATE vs. UML	0.05621	1	1	0.90429	0.57813	0.69141
ARCHIMATE vs. xADL	0.01377	0.62333	0.16410	0.86914	0.77344	0.82031
ARCHIMATE vs. Y_DON	0.59196	1	0.02271	0	0.25195	0.06055
EAST-ADL vs. MIND	$7.2x10^{-06}$	1	$4.9x10^{-08}$	0.17578	0.24804	0
EAST-ADL vs. MODELICA	1	1	1	0.39844	0.39844	0.05664
EAST-ADL vs. PCM	0.03129	$5.4x10^{-05}$	$2.6x10^{-08}$	0.19531	0.14453	0
EAST-ADL vs. SDL	0.31385	1	$7.0x10^{-08}$	0.35156	0.35156	0
EAST-ADL vs. UML	0.06930	0.01916	$6.6x10^{-06}$	0.22852	0.14844	0
EAST-ADL vs. xADL	0.19533	0.22138	0.00681	0.22461	0.19531	0.00977
EAST-ADL vs. Y_DON	$6.4x10^{-13}$	$1.2x10^{-07}$	$\ll 2.2x10^{-16}$	0	0	0
MIND vs. MODELICA	$1.8x10^{-06}$	1	$1.8x10^{-05}$	0.78516	0.69336	1
MIND vs. PCM	0.70166	0.04030	1	0.72656	0.21289	0.44922
MIND vs. SDL	0.10055	1	1	0.65625	0.52734	0.52734
MIND vs. UML	0.42606	1	1	0.72656	0.21289	0.63672
MIND vs. xADL	0.15858	1	0.22062	0.72656	0.30859	0.82031
MIND vs. Y_DON	0.08503	0.00038	0.01623	0.34375	0	0
MODELICA vs. PCM	0.01237	0.00024	$1.1x10^{-05}$	0.17188	0.17188	0
MODELICA vs. SDL	0.15858	1	$2.5x10^{-05}$	0.42969	0.42969	0
MODELICA vs. UML	0.02836	0.05604	0.00117	0.17188	0.17188	0.00391
MODELICA vs. xADL	0.10055	0.52070	0.26365	0.17188	0.17578	0.07227
MODELICA vs. Y_DON	$1.1x10^{-13}$	$7.0x10^{-07}$	$1.1x10^{-13}$	0	0	0
PCM vs. SDL	1	0.07003	1	0.54688	0.73047	0.50391
PCM vs. UML	1	1	1	0.63867	0.58789	0.66211
PCM vs. xADL	1	0.63987	0.16231	0.61914	0.70898	0.83008
PCM vs. Y_DON	$3.2x10^{-05}$	1	0.02318	0	0.33789	0.01172
SDL vs. UML	1	1	1	0.45313	0.28125	0.60938
SDL vs. xADL	1	1	0.26100	0.45313	0.41797	0.76563
SDL vs. Y_DON	$5.2x10^{-07}$	0.00082	0.01322	0.18750	0.05078	0.05859
UML vs. xADL	1	1	1	0.46484	0.70117	0.73828
UML vs. Y_DON	$1.0x10^{-05}$	0.22630	0.00040	0	0.11133	0
xADL vs. Y_DON	$1.3x10^{-06}$	0.01990	$1.8x10^{-07}$	0	0.05078	0



	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7	Req 8	Req 9	Req 10	Req 11	Req 12	Req 13	Req 14	Req 15	Req 16
AADL	10	8	6	10	7	7	8	5	8	8	6	6	6	6	5	4
ACME	6	9	7	10	7	8	9	6	9	9	7	7	7	7	6	5
ARCHIMATE	15	18	15	17	15	17	18	14	18	18	16	16	16	16	15	14
EAST-ADL	9	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
MIND	78	113	111	143	104	110	111	89	116	160	108	91	83	93	82	74
MODELICA	27	35	1	35	30	32	35	27	33	35	29	31	31	31	28	24
PCM	4	7	4	5	4	6	7	3	7	8	5	5	5	5	4	3
SDL	5	8	6	9	5	7	8	4	7	8	6	6	6	6	5	4
UML	3	6	4	5	5	5	7	3	5	8	4	4	4	3	3	4
xADL	13	16	12	15	13	13	16	12	13	14	12	12	12	12	11	10
Y_DON	18	18	20	18	20	21	17	21	23	19	19	19	19	18	20	18

FIGURE 6 Number of Architectural Language Model Elements for each requirement

5 | DISCUSSION

There is a substantial inherent gap between requirement descriptions and ALs, because the transformation from requirements to architecture models is not included in the model-driven architecture life cycle, which starts from an analysis model (or design model) and ends with deployed code[54]. The reason for this exclusion is perhaps that requirements are always written with natural language texts, which are not a model formal enough to be understood by computers[55]. As a result, natural language requirements are not suitable for automated transformations.

Each AL is characterized by distinct conceptual architectural elements to satisfy different stakeholder concerns, giving engineers different concepts in order to specify a software architecture: AADL (*ComponentImpl*, *Connector*, *Block*); Acme (*System*, *Component*, *Port*); Archimate (*Application collaboration*, *Business process*, *Infrastructure interface*); EAST-ADL (*EventChain*, *ErrorBehavior*, *LifecycleStageKind*); MIND (*Class*, *Operation*, *Property*); Modelica (*class*, *equation*, *connect*); PCM (*Entity*, *Role*, *RepositoryComponent*); SDL (*SdlNamedElement*, *SdlFeature*, *SdlAgent*); UML (*Constraint*, *Actor*, *Lifeline*); xADL (*Link*, *Point*, *Interface*); and Y_DON (*Process*, *Data Store*, *Entity*).

If we classify each AL according to the concepts used in its specification, there is a great disparity among ALs. For instance, ACME uses seven concepts to specify its architecture, Y_DON uses four concepts to specify its architecture, and SDL uses eleven concepts to specify its architecture. In addition, different ALs use a high number of concepts (more than one hundred concepts) to specify their architectures. For example, UML uses around 110 concepts, AADL uses more than 200 concepts, and EAST-ADL uses more than 400 concepts. We analyzed the results looking for correlations between requirements traceability performance and the size of concepts in ALs specifications, but we did not find any correlations.

The concepts of each AL specification influence the number of elements that an architecture model contains (see Figure 6). However, the number of model elements does not offer any explanation for the results achieved. Taking into account the same requirements for all ALs, some ALs like PCM or SDL use around five elements to model a specific requirement, while other

ALs such as MIND use more than one hundred elements to model the same requirement. This is the reason why we believe that MIND obtains good precision values (69% of the elements found belong to the requirement), but low recall values (only 4% of the elements of the requirement are found). The approach has to find more than 100 elements in a model with more than 670 elements. In the case of Y_DON, which obtains the best precision value (100%), the size of the requirements is around 19 elements on average. It also obtains good values for recall (69%). In this case, the approach has to find 19 elements in models of about 60 elements. Nevertheless, the analysis of our results did not reveal a correlation between the number of model elements and the requirements traceability results achieved. Neither the AL concepts nor the requirement model elements explain the performance differences among ALs. However, we realize that the architectural languages that allow the specification of the requirements closer to the domain due to the architectural elements used obtain better results in precision and recall. This is the case of Y_DON, which we consider the most abstract language of those studied because it allows defining the requirements in the closest way to the domain. On the opposite side, we find MODELICA, which obtains the worst results because the final models of the requirements are very far from the domain, it is the least abstract language.

ALs can be classified according to different criteria[56], [16]. One of the well-accepted AL classifications is those presented by Taylor et al.[16], which distinguishes four AL categories based on the engineers' concerns and purposes. The authors propose four categories: 1) General-Purpose/Research Languages (e.g., UML or Archimate); 2) Early Architecture Description Languages (e.g., PCM); 3) Domain-and Style-specific Languages (e.g., AADL or Modelica); and 4) Extensible architecture Description Languages (e.g., xADL or Acme). Note that these categories are not mutually exclusive. For example, this paper classifies UML and AADL as General-Purpose/Research Languages. However, both ALs can be considered Extensible Architecture Description Languages also. UML uses extension mechanisms through profiles and stereotypes and AADL uses extension mechanisms through annex libraries[57]. Even though different ALs can be classified in different categories, in that case, the existing works in the field classify both UML and AADL as General-Purpose/Research Languages[58,57,16,59,60]. For this reason, we followed the same classification[16], classifying both UML and AADL ALs as General-Purpose/Research Languages.

Taking this classification into account, our analysis shows that General-Purpose/Research Languages achieve the best results during requirements traceability followed by Extensible Architecture Description Languages. In contrast, both Early Architecture Description Languages and Domain-and Style-specific Languages achieve worse results during requirements traceability. The architecture models specified using those ALs classified as General-Purpose/Research Languages and those ALs classified as Extensible architecture Description Languages contain more generic and domain-independent terms. In contrast, the architecture models specified using those ALs classified as Early Architecture Description Languages and those ALs classified as Domain- and style-specific Languages contain terms that are more closely aligned (i.e., high degree of textual similarity) with domain-specific terms.

One might think that using domain-specific terms instead of generic terms would help during the requirements traceability activity since requirements are made up of domain terms. However, counter-intuitively, our work reveals that it has exactly the opposite effect. Those ALs use more domain-specific terms during architecture model specification (Early Architecture Description Languages and Domain-and Style-specific Languages) leading to an excessive number of domain terms in the architecture models, hindering requirements traceability. In contrast, ALs that enable engineers to use more generic and domain-independent terms to specify their architectures (General-Purpose/Research Languages and Extensible architecture Description Languages) lead to more effective use of domain-specific terms, resulting in better results during requirements traceability.

Furthermore, we have noted that the results of TLR are closely aligned with ALs' grammar constraints. In some cases, ALs' grammar constraints are hindering requirements traceability with architecture models. In this paper, we consider grammar constraints such as those barriers that ALs' specifications apply to specify natural language from requirements. This is because some ALs do not enable designers to specify open compound words from natural language requirements in architecture models. Some ALs, such as Y_DON, MIND, or ARCHIMATE (those that best results achieve) enable designers to define the term with spaces and special characters. Nevertheless, other ALs such as MODELICA, EAST-ADL, or ACME (those that achieve the worst results) replace domain-specific terms' spaces with a low bar, or directly remove the empty space by joining the words which conform that term.

In order to illustrate this, we use the following requirement from a real-world train: "The PLC will disable the order of 'pantograph equipment's connection' if the state of the knife switch is unknown, with the train in shutdown sequence". In that case, 'equipment' is a domain-specific term that plays in a variety of industrial domains such as the aeronautic, automotive, medical, nuclear, and railway domains, as well as many more. In addition, the domain-specific term 'knife switch' is a compound word that can be found in different ways depending on the architecture model, such as "knife switch", "knife_switch", or "knifeswitch".

We strongly recommend researchers and practitioners take that into consideration, defining and considering a list of those terms which can be simultaneously present in both domain-specific languages as in architecture model specification concepts. Another recommendation perhaps less obvious but no less important is the fact that some ALs have grammatical constraints which hinder requirement traceability with architecture models. For this reason, we recommend refining the initial architecture models through patterns to achieve the change of requirements and traceability links. AL designers should not use words that are domain-specific terms and architecture model specification concepts at the same time. In addition, AL designers should refine the generated initial architecture models in order to delete AL constraints such as low bars or others.

Our results can help AL designers by providing them with which type of words play the same role on domain-specific terms and on architecture model specification concepts (e.g., 'equipment', 'door', or 'signal receptor', among others) in a way that generates noise in the architecture models and that hampers the requirement traceability performance. Furthermore, our results are relevant to researchers and practitioners by providing them with information about the differences among ALs (e.g., the number of architecture definition concepts required to specify a particular model element in different languages). It enables practitioners to make more informed decisions about ALs and to choose the one that best fits their needs in terms of requirements traceability.

6 | THREATS TO VALIDITY

In this section, we use the classification of threats to validity of[61,62] to acknowledge the limitations of our work:

1. **Construct Validity:** This aspect of validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind. In order to minimize this risk, we use objective and widely accepted measures (Precision, Recall, F-Measure, Matthews Correlation Coefficient), which have been used before by other researchers in the community[63]. Moreover, we performed a fair comparison among the architectural languages. We executed an independent run for each of the 16 requirements for each of the 11 ALs considered for this study.

2. **Internal Validity:** This aspect of validity is of concern when causal relations are examined. There is a risk that the factor being investigated may be affected by other neglected factors. The choice of the k value in the application of SVD can produce sub-optimal accuracy when using LSI for software artifacts[64]. In order not to affect the comparison, we use the same k value when performing requirement traceability in each of the architecture languages. Although evaluating the influence of the k value could be relevant, it is out of the scope of this study. In addition, we take into account reliability. This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. The requirements and architecture models of the trains used in our experiment were provided by our industrial partner's engineers, as well as the domain terms, which were crafted by domain experts who were not involved in this research. Furthermore, the experience in the use of Architectural Languages (ALs) by a system architect modeler can impact the results. All of the system architects of our industrial partner have more than 15 years of experience in the field. In order to mitigate this problem, all of the system architects have reviewed all of the models. That is, each system architect verifies that the models constructed by the rest of the team are well constructed and a good representation of the requirement.

3. **External Validity:** This aspect of validity is concerned with to what extent it is possible to generalize the findings and to what extent the findings are of relevance for other cases. The architectural languages used in our research to model the real-world CAF architecture are a diverse set of ALs used in the industry that can be applied to different domains. In addition, the RTAL approach does not rely on the specific conditions of any domain. Nevertheless, our results should be replicated with other case studies before assuring their generalization.

4. **Conclusion Validity:** This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers. To avoid this threat, all of the inputs were provided by our industrial partner. Moreover, we used precision, recall, F-measure, and MCC metrics to analyze the confusion matrix obtained in the evaluation. We also employed standard statistical analysis following accepted guidelines[50] (Quade test, Holm's post hoc analysis, and Vargha and Delaney's $\hat{A}_{12}$).

TABLE 4 Related Work Overview

	From Requirements to	Languages used	Industrial Evaluation
Eaddy et al.[71]	Code	C#	NO
Shahid et al.[65]	Code	Not specified	NO
Al-Saiyd et al.[72]	Architecture	UML	NO
Zisman et al.[66]	Code	UML	Yes (Philips)
Sherba et al.[67]	Architecture	Not specified	NO
Abbors et al.[68]	Test cases	UML, QML	NO
Delater et al.[69]	Code	UML, Java	NO
Han, J.[73]	Architecture	HTML	Yes (NATS)
Cleland et al.[74]	Performance models	Java	NO
Bouquet et al.[70]	Test cases	Not Specified	Yes (Smart Card Industry)
This Work	Architecture	11 ALs	Yes (CAF)

7 | RELATED WORK

Requirements traceability plays an important role in the Software Engineering Community[65,66,67,68,69,70]. In this section, we summarize related works and compare this study with them with regard to the domain of Software Artifact Traceability. Table 4 presents a review of related works to requirements traceability. These works are classified in terms of target artifacts, languages used, and industrial evaluation. First, we introduce the state-of-the-art of traceability among requirements and code, second, we describe the state-of-the-art of traceability among requirements and architectures. Finally, we present the state-of-the-art of traceability among requirements and other software artifacts such as test cases or performance models.

Some works focus on the traceability between requirements and source code. For instance, Eaddy et al.[71] present a systematic methodology for identifying which code is related to which requirement and a suite of metrics for quantifying the amount of crosscutting code. In[65], the authors present an approach for locating the traceability of functional requirements into artifacts such as methods, classes, and packages. Delater et al.[69] present an approach for tracing requirements and source code during software development to satisfy the information needs of developers regarding requirements during development. Zisman et al.[66] automate the generation of traceability relations between textual requirement artifacts and object models using heuristic rules. However, these approaches deal with the traceability between source code and requirements. In contrast, our work recovers the traceability between requirements and architecture models.

Other works address traceability among requirements and architectures. For instance, Al-Saiyd et.al[72] describe the impact of changing the requirements in the architectural software design based on risks and the corresponding affected areas of the development systems. They explore the impacts of new or changing system requirements on existing and future system goals and identify factors that may influence the software architecture design. Sherba et al.[67] proposed an approach, TraceM, that is based on techniques from open-hypermedia and information integration. TraceM manages traceability links between requirements and architecture. TraceM enables the creation, maintenance, and viewing of traceability relationships in tools that software professionals use on a daily basis. In[73], the author proposes a tool for managing system requirements, system architectures, and the traceability between them. The tool involves an underlying information model that captures the key concepts and relationships of requirements engineering and architecture design. In contrast to these works, our work analyzes the influence of different Architectural Languages on RTAL, considering a set of eleven ALs used by industry.

Finally, some works deal with recovering traceability links among requirements and other software artifacts. In[68], the authors present an approach for tracing product requirements across a model-based testing process, from informal documents via test models to test cases, and back to requirements and test models. Cleland et al.[74] address traceability by proposing a method for establishing and utilizing traceability links between requirements and performance models. The approach identifies where relationships exist between requirements and performance models and supports the process of analyzing the impact of a proposed change upon the performance of the system through dynamic re-execution of requirement-dependent models. In[70], an approach to automatically produce the traceability matrix from requirements to test cases is presented. In contrast, we evaluate the influence of using the most popular ALs on RTAL, measuring the results based on four performance indicators, which include Precision, Recall, F-measure, and Matthews Correlation Coefficient. Finally, we conclude by providing insights about how traceability links recovery can be improved in requirements and architecture models.

8 | CONCLUSION

Many Architectural Languages (ALs) can be found today[4], each of which has the chief aim of becoming the ideal language for specifying software systems architectures. In industrial scenarios, it is common to use different ALs to specify different software systems. However, despite the popularity of different ALs, the question of how ALs influence software system maintainability has not yet received much attention.

Motivated by this challenge, we have analyzed the influence of the ALs used by industry[4] in one of the most commonly performed activities during the software system maintenance phase: requirements traceability. Actively supporting traceability is critical to the software engineering community in order to verify and trace non-reliable parts[9] and to decrease the expected defect rate in development software[10].

We conducted an evaluation in the railway domain with our industrial partner CAF, a worldwide leader in railway manufacturing. Requirements Traceability to Architecture Languages (RTAL) achieves the best results when the AL used is a General-Purpose/Research Language. The next best results are achieved by those ALs that are classified as Extensible Architecture Description Languages. Those ALs that use more generic and domain-independent terms to specify their architectures obtain better results during requirements traceability.

Our results can help AL designers by providing them with which type of terms play the same role on domain-specific and on architecture model specifications in a way that generates noise in the architecture models and that hampers the requirement traceability performance. Furthermore, our results are relevant to researchers and practitioners by providing them with information about the differences among ALs enabling practitioners to make more informed decisions about ALs and to choose the one that best fits their needs in terms of requirements traceability. As part of our future work, we are planning to explore the influence of ALs on bug location.

ACKNOWLEDGMENTS

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

References

1. Martin JN. Overview of the Revised Standard on Architecture Description–ISO/IEC 42010. In: . 31. Wiley Online Library. ; 2021: 1363–1376.

2. Grau A, Shihada B, Soliman M. Architectural Description Languages and their Role in Component Based Design. *Project Report, Department of Computer Science, University of Waterloo, Canada, Available: http://www. cs. uwaterloo. ca/˜ bshihada/adl. pdf* 2002.

3. Dashofy EM, Hoek Avd, Taylor RN. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2005; 14(2): 199–245.

4. Malavolta I, Lago P, Muccini H, Pelliccione P, Tang A. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering* 2013; 39(6): 869–891. doi: 10.1109/TSE.2012.74

5. Medvidovic N. Moving architectural description from under the technology lamppost. *Proceedings - 32nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA* 2006: 2–3. doi: 10.1109/EUROMICRO.2006.47

6. Tian F, Wang T, Liang P, Wang C, Khan AA, Babar MA. The Impact of Traceability on Software Maintenance and Evolution: A Mapping Study. *J. Softw. Evol. Process* 2021; 33(10). doi: 10.1002/smr.2374

7. Gotel OCZ, Finkelstein ACW. An Analysis of the Requirements Traceability Problem. *1st International Conference on Requirements Engineering (RE 1994)* 1994: 94–101. doi: 10.1109/ICRE.1994.292398

8. Spanoudakis G, Zisman A. Software Traceability: a Roadmap. *Handbook Of Software Engineering And Knowledge Engineering* 2005; III: 395–428. doi: 10.1142/9789812775245_0014

9. Watkins R, Neal M. Why and how of requirements tracing. *IEEE Software* 1994; 11(4): 104–106. doi: 10.1109/52.300100

10. Rempel P, Mader P. Preventing defects: The impact of requirements traceability completeness on software quality. *IEEE Transactions on Software Engineering* 2017; 43(8): 777–797. doi: 10.1109/TSE.2016.2622264

11. Landauer T, Foltz P, Laham D. An introduction to latent semantic analysis. *Discourse Processes* 1998; 25(October): 259–284. doi: 10.1080/01638539809545028

12. Rubin J, Chechik M. A survey of feature location techniques. *Domain Engineering: Product Lines, Languages, and Conceptual Models* 2013: 29–58. doi: 10.1007/978-3-642-36654-3_2

13. Wong WE, Gao R, Li Y, et al. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 2016; 42(8): 707–740.

14. Salton G, McGill MJ. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc. . 1986.

15. Marcus A, Sergeyev A, Rajlich V, Maletic JIJ. An Information Retrieval Approach to Concept Location in Source Code. *Wcre 2004* 2004: 214–223. doi: 10.1109/WCRE.2004.10

16. Medvidovic N, Taylor RN. Software architecture: foundations, theory, and practice. In: . 2. IEEE. ; 2010: 471–472.

17. Feiler P. The Open Source AADL Tool Environment (OSATE). tech. rep., Carnegie Mellon University Software Engineering Institute Pittsburgh United . . . ; : 2019.

18. Schmerl B, Garlan D. AcmeStudio: supporting style-centered architecture development. In: ; 2004: 704-705.

19. Beauvoir P, Sarrodie JB. Archi-Open Source Archimate Modelling.; 2019.

20. Wallnau KC, Ivers J. Snapshot of CCL: A language for predictable assembly. tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST; : 2003.

21. Magee J, Dulay N, Eisenbach S, Kramer J. Specifying distributed software architectures. In: Springer. ; 1995: 137–153.

22. Blom H, Lönn H, Hagl F, et al. EAST-ADL: An architecture description language for automotive software-intensive systems. In: IGI Global. 2013 (pp. 456–470).

23. Carmichael AR. Defining software architectures using the hierarchical object-oriented design method (HOOD). In: ; 1992: 211–219.

24. Van't Wout J, Waage M, Hartman H, Stahlecker M, Hofman A. *The integrated architecture framework explained: why, what, how*. Springer Science & Business Media . 2010.

25. Bettin J, Cook W, Clark T, Kelly S. Knowledge industry survival strategy (KISS) fundamental principles and interoperability requirements for domain specific modeling languages. In: ; 2009: 709–710.

26. Van Ommering R, Van Der Linden F, Kramer J, Magee J. The Koala component model for consumer electronics software. *Computer* 2000; 33(3): 78–85.

27. Consortium O, others . The MIND project, Jan. 2013. *URL http://mind. ow2. org/.[Online* 2014.

28. Fritzson P, Engelson V. Modelica—A unified object-oriented language for system modeling and simulation. In: Springer. ; 1998: 67–90.

29. Bellissard L, De Palma N, Féliot D. The olan architecture definition language. *C3DS Technical Report* 2000; 24.

30. Reussner R, Becker S, Burger E, et al. The Palladio component model. 2011.

31. Luckham DC, Kenney JJ, Augustin LM, Vera J, Bryan D, Mann W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 1995; 21(4): 336–354.

32. Becker S, Bulej L, Bures T, et al. Q-ImPrESS project deliverable D2. 1: service architecture meta model (SAMM). *Project Deliverable* 2008.

33. Deltour J, Faivre A, Gaudin E, Lapitre A. Model-based testing: an approach with SDL/RTDS and DIVERSITY. In: Springer. ; 2014: 198–206.

34. Schriber TJ, Brunner DT, Smith JS. Inside discrete-event simulation software: how it works and why it matters. In: IEEE. ; 2013: 424–438.

35. Booch G. *The unified modeling language user guide*. Pearson Education India . 2005.

36. Dashofy E, Asuncion H, Hendrickson S, Suryanarayana G, Georgas J, Taylor R. Archstudio 4: An architecture-based meta-modeling environment. In: IEEE. ; 2007: 67–68.

37. Feiler PH, Lewis B, Vestal S. The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. In: ; 2003: 1–10.

38. University. CM. Acme.; 1998.

39. Lankhorst M, Proper H, Jonkers H. The Anatomy of the ArchiMate Language. *International Journal of Information System Modeling and Design* 2010; 1(1): 1–32. doi: 10.4018/jismd.2010092301

40. Debruyne V, Simonot-Lion F, Trinquet Y. EAST-ADL — An Architecture Description Language. In: Dissaux P, Filali-Amine M, Michel P, Vernadat F., eds. *Architecture Description Languages*Springer US; 2005; Boston, MA: 181–195.

41. Belina F, Hogrefe D. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems* 1989; 16(4): 311 - 341. doi: https://doi.org/10.1016/0169-7552(89)90078-0

42. Khare R, Guntersdorfer M, Oreizy P, Medvidovic N, Taylor RN. xADL: enabling architecture-centric tool integration with XML. In: ; 2001: 9.

43. Yourdon E. *Modern systems analysis*. Prentice-Hall . 1989.

44. Oliveto R, Gethers M, Poshyvanyk D, De Lucia A. On the equivalence of information retrieval methods for automated traceability link recovery. *IEEE International Conference on Program Comprehension* 2010: 68–71. doi: 10.1109/ICPC.2010.20

45. Antoniol , Canfora , Casazza , De Lucia . Information retrieval models for recovering traceability links between code and documentation. *Proceedings International Conference on Software Maintenance ICSM-94* 2000: 40–49. doi: 10.1109/ICSM.2000.883003

46. Hofmann T. Probabilistic latent semantic indexing. *In Proceedings of ACM SIGIR* 1999.

47. Eyal-salman H, Seriai Ad, Dony C, et al. Feature Location in a Collection of Product Variants : Combining Information Retrieval and Hierarchical Clustering. *The 26th International Conference on SoftwareSoftware Engineering and Knowledge Engineering (SEKE 2014)* 2014: 426–430.

48. Apache OpenNLP - a machine learning based toolkit for the processing of natural language text.; 2010.

49. Abeles P. Efficient Java Matrix Library Contents.; 2010.

50. Arcuri A, Briand L. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing Verification and Reliability* 2014; 24(3): 219–250.

51. García S, Fernández A, Luengo J, Herrera F. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences* 2010; 180(10). doi: 10.1016/j.ins.2009.12.010

52. Vargha A, Delaney HD. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 2000; 25(2): 101-132. doi: 10.3102/10769986025002101

53. Grissom RJ, Kim JJ. *Effect sizes for research: A broad practical approach.* Mahwah, NJ: Earlbaum . 2005.

54. Yue T, Briand L. A systematic review of transformation approaches between user requirements and analysis models. *Requir. Eng.* 2011; 16. doi: 10.1007/s00766-010-0111-y

55. Wang F, Yang ZB, Huang ZQ, et al. An Approach to Generate the Traceability Between Restricted Natural Language Requirements and AADL Models. *IEEE Transactions on Reliability* 2020; 69(1): 154-173. doi: 10.1109/TR.2019.2936072

56. Mishra P, Dutt N. Architecture description languages for programmable embedded systems. *IEE proceedings-computers and digital techniques* 2005; 152(3): 285–297.

57. Ozkaya M. The analysis of architectural languages for the needs of practitioners. *Software: Practice and Experience* 2018; 48(5): 985–1018.

58. Lago P, Malavolta I, Muccini H, Pelliccione P, Tang A. The Road Ahead for Architectural Languages. *IEEE Software* 2014; prePrint: 1. doi: 10.1109/MS.2014.28

59. Ozkaya M. Analysing UML-based software modelling languages. *Journal of Aeronautics and Space Technologies* 2018; 11(2): 119–134.

60. Amjad A, Haq SU, Abbas M, Arif MH. UML Profile for Business Process Modeling Notation. In: IEEE. ; 2021: 389–394.

61. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Sofware Engineering, Empirical* 2008; 14: 131–164. doi: 10.1007/s10664-008-9102-8

62. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. Experimentation in software engineering. *Experimentation in Software Engineering* 2012; 9783642290: 1–236. doi: 10.1007/978-3-642-29044-2

63. Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T. Automatic query reformulations for text retrieval in software engineering. *Proceedings - International Conference on Software Engineering* 2013: 842–851. doi: 10.1109/ICSE.2013.6606630

64. Panichella A, Dit B, Oliveto R, Penta MD, Poshyvanyk D, Lucia AD. Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 2016: 314–325. doi: 10.1109/SANER.2016.97

65. Shahid M, Ibrahim S. A New Model For Requirements to Code Traceability to Support Code Coverage Analysis. *Asian Academic Research Journal of Multidisciplinary (AARJMD)* 2013; 1(14): 159–172.

66. Zisman A, Spanoudakis G, Pérez-Miñana E, Krause P. Tracing Software Requirements Artefacts. In: ; 2003: 448–455.

67. Sherba S, Anderson K. A framework for managing traceability relationships between requirements and architectures. *STRAW'03 Second International SofTware Requirements to Architectures Workshop* 2003: 150.

68. Abbors F, Truşcan D, Lilius J. Tracing requirements in A model-based testing approach. *1st International Conference on Advances in System Testing and Validation Lifecycle, VALID 2009* 2009: 123–128. doi: 10.1109/VALID.2009.15

69. Delater A, Paech B. Tracing Requirements and Source Code during Software Development: An Empirical Study. *ESEM 2013: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* 2013: 25–34. doi: 10.1109/ESEM.2013.16

70. Bouquet F, Jaffuel E, Legeard B, Peureux F, Utting M, New-zealand H. Requirements Traceability in Automated Test Generation - Application to Smart Card Software Validation. *SIGSOFT Softw. Eng. Notes* 2005; 30(4): 1–7. doi: 10.1145/1083274.1083282

71. Eaddy M, Aho A, Murphy GC. Identifying, assigning, and quantifying crosscutting concerns. *Proceedings - ICSE 2007 Workshops: First International Workshop on Assessment of Contemporary Modularization Techniques, ACoM'07* 2007. doi: 10.1109/ACOM.2007.4

72. Al-Saiyd N, Zriqat E. Analyzing the Impact of Requirement Changing on Software Design. *European Journal of Scientific Research* 2015; 136.

73. Han J. TRAM: A tool for requirements and architecture management. *Proceedings - 24th Australasian Computer Science Conference, ACSC 2001* 2001: 60–68. doi: 10.1109/ACSC.2001.906624

74. Cleland-Huang J, Chang CK, Sethi G, Javvaji K, Hu H, Xia J. Automating speculative queries through event-based requirements traceability. *Proceedings of the IEEE International Conference on Requirements Engineering* 2002; 2002-Janua: 289–296. doi: 10.1109/ICRE.2002.1048540

APPENDIX

A CHARTS WITH THE PRECISION AND RECALL RESULTS FOR EACH REQUIREMENT FOR OUR REAL-WORLD CASE STUDY AND THE ELEVEN ALS

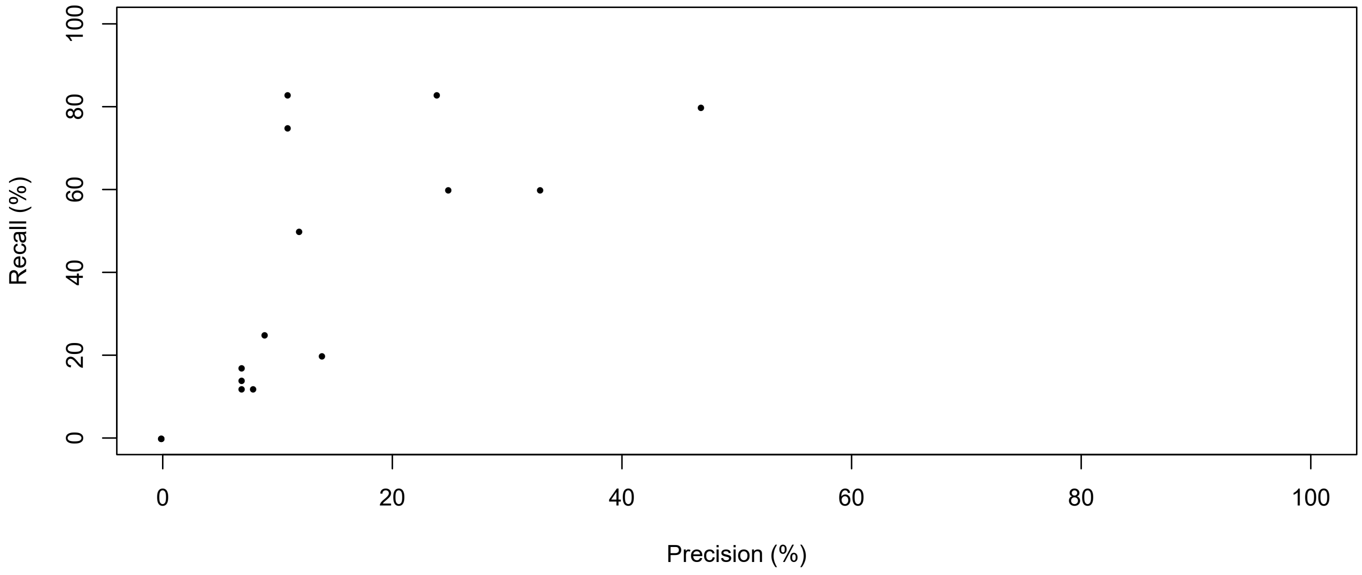


FIGURE A1 Mean Precision and Recall for the CAF Case Study for AADL

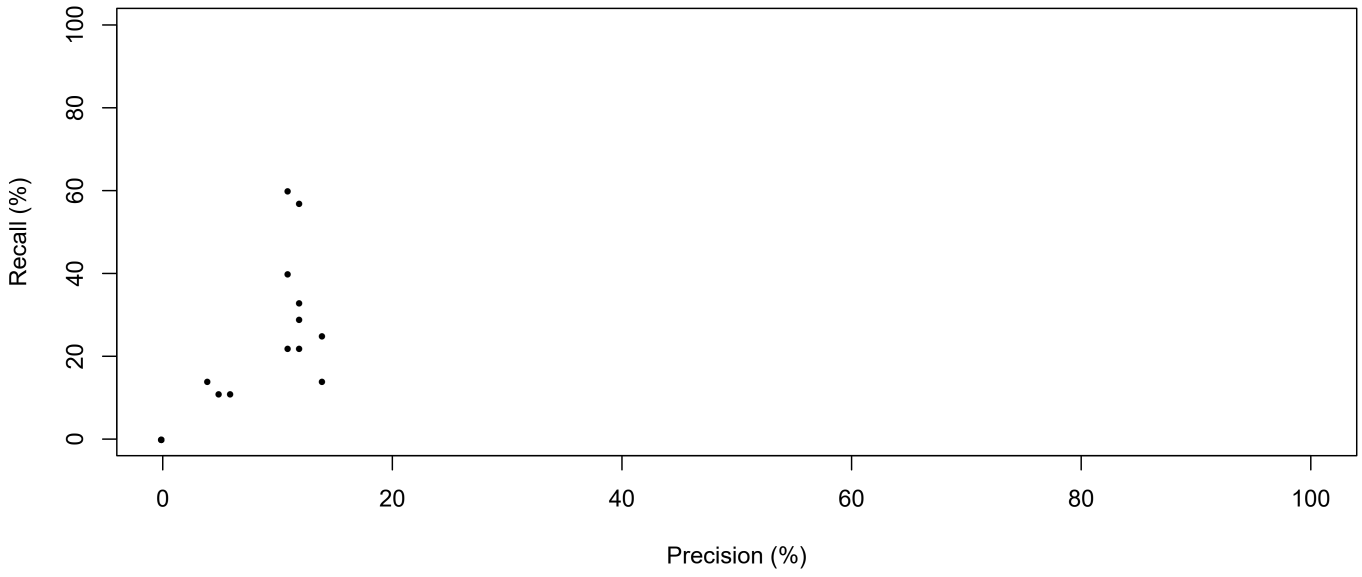


FIGURE A2 Mean Precision and Recall for the CAF Case Study for ACME

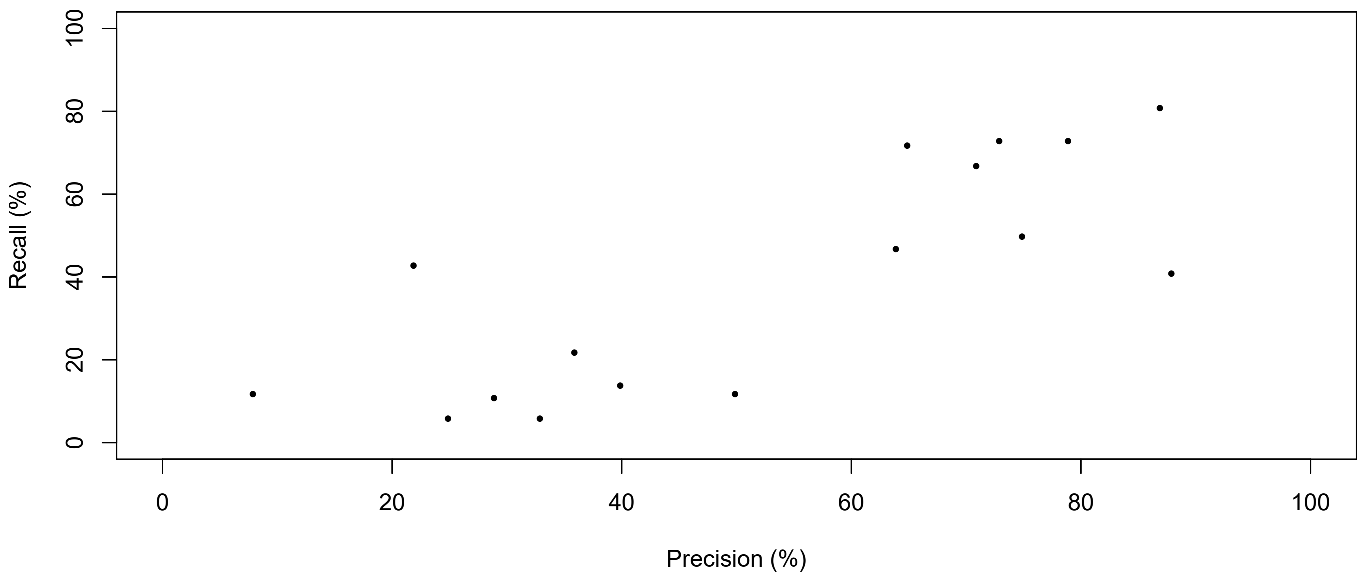


FIGURE A3 Mean Precision and Recall for the CAF Case Study for ARCHIMATE

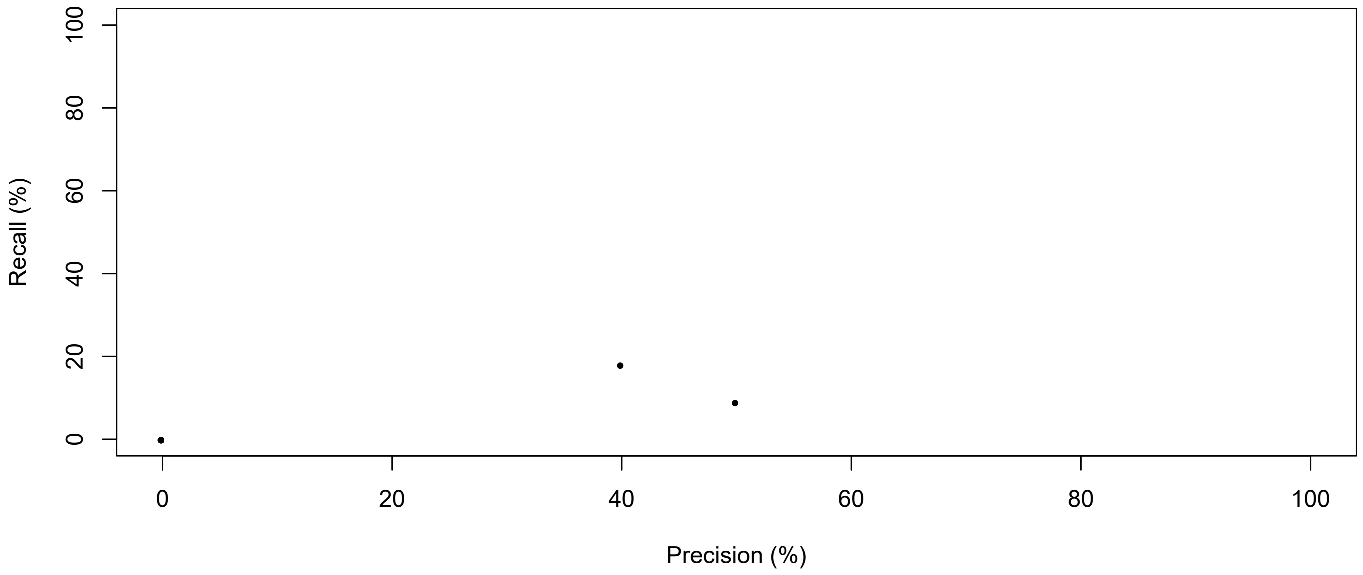


FIGURE A4 Mean Precision and Recall for the CAF Case Study for EAST-ADL

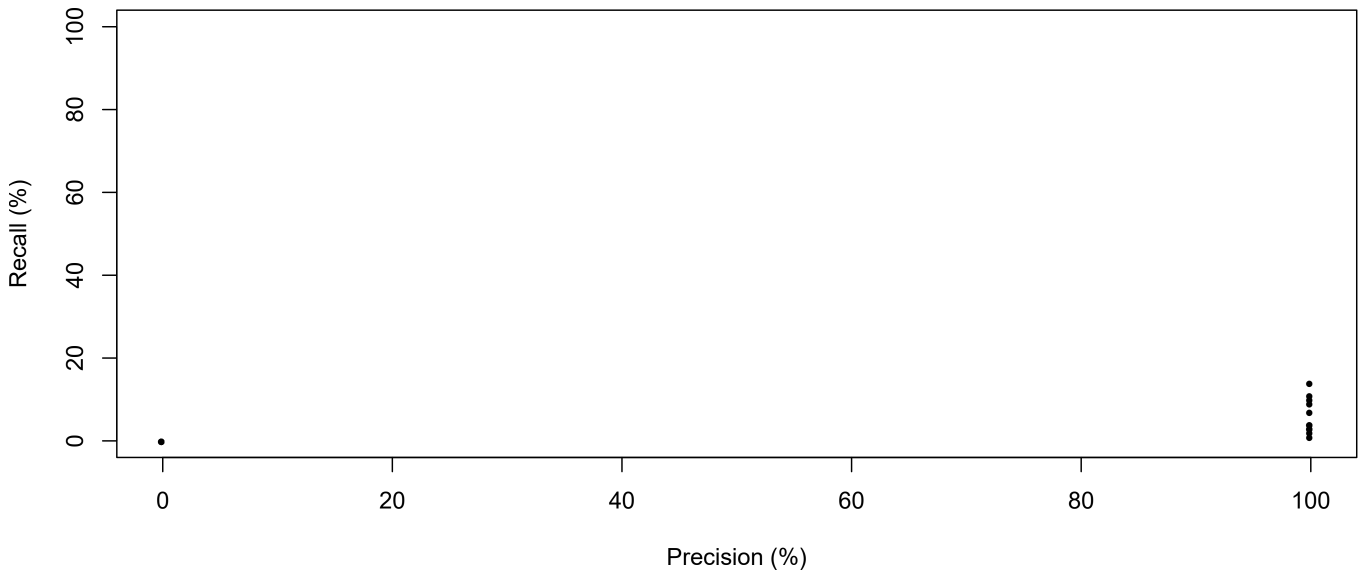


FIGURE A5 Mean Precision and Recall for the CAF Case Study for MIND

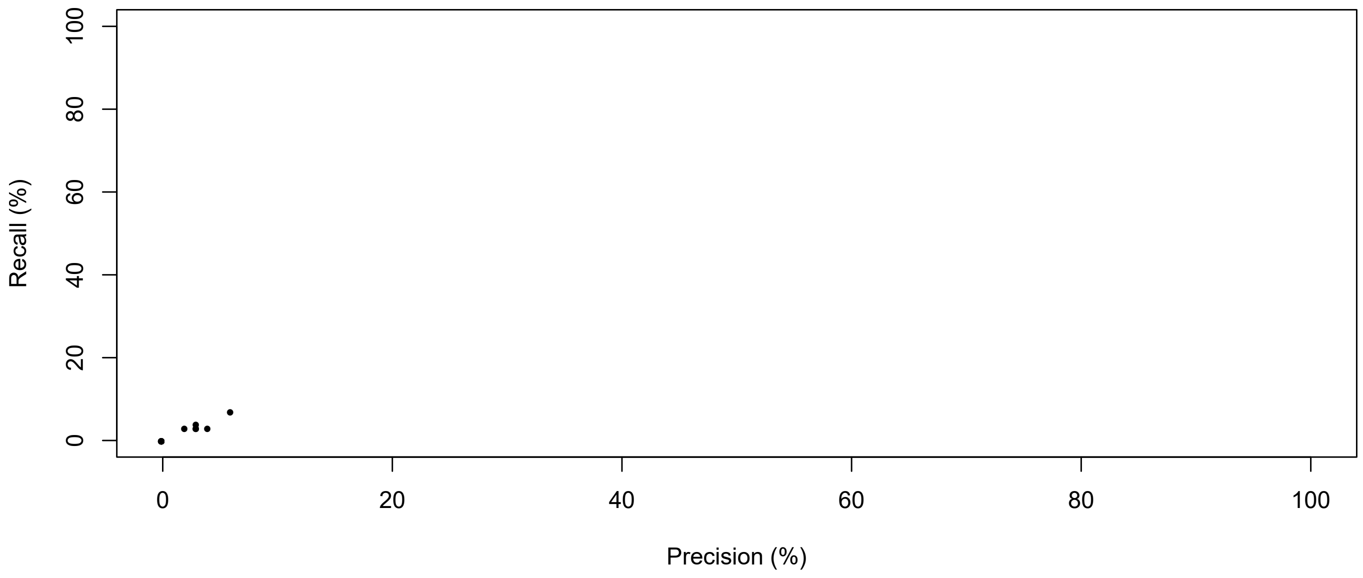


FIGURE A6 Mean Precision and Recall for the CAF Case Study for MODELICA

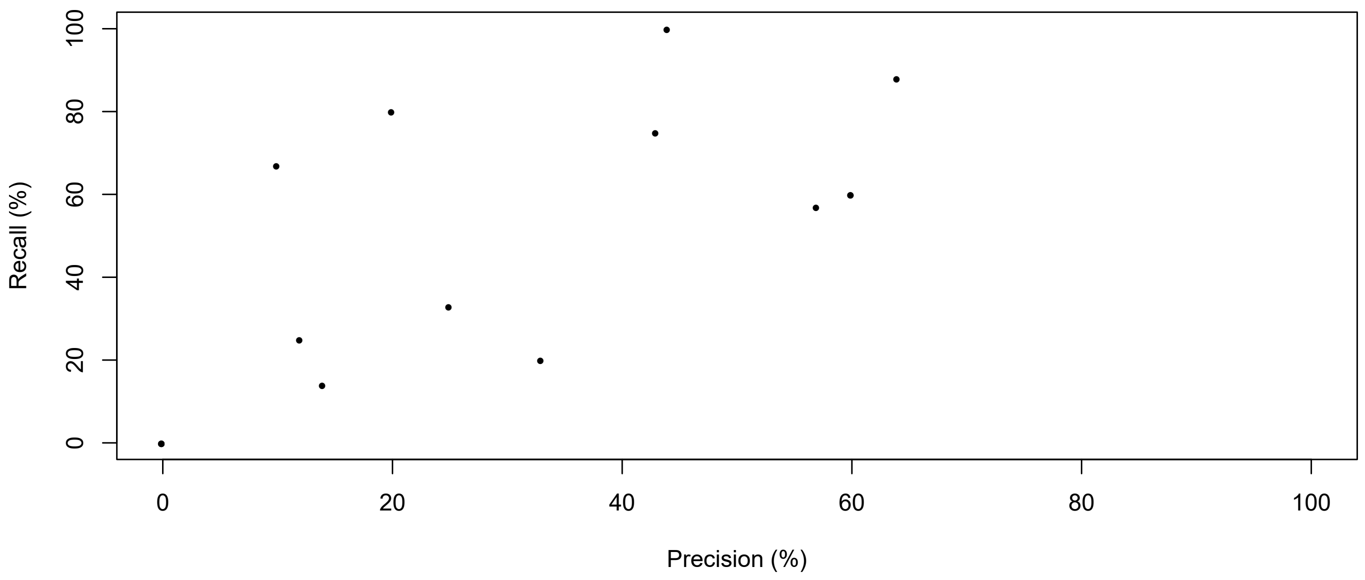


FIGURE A7 Mean Precision and Recall for the CAF Case Study for PCM

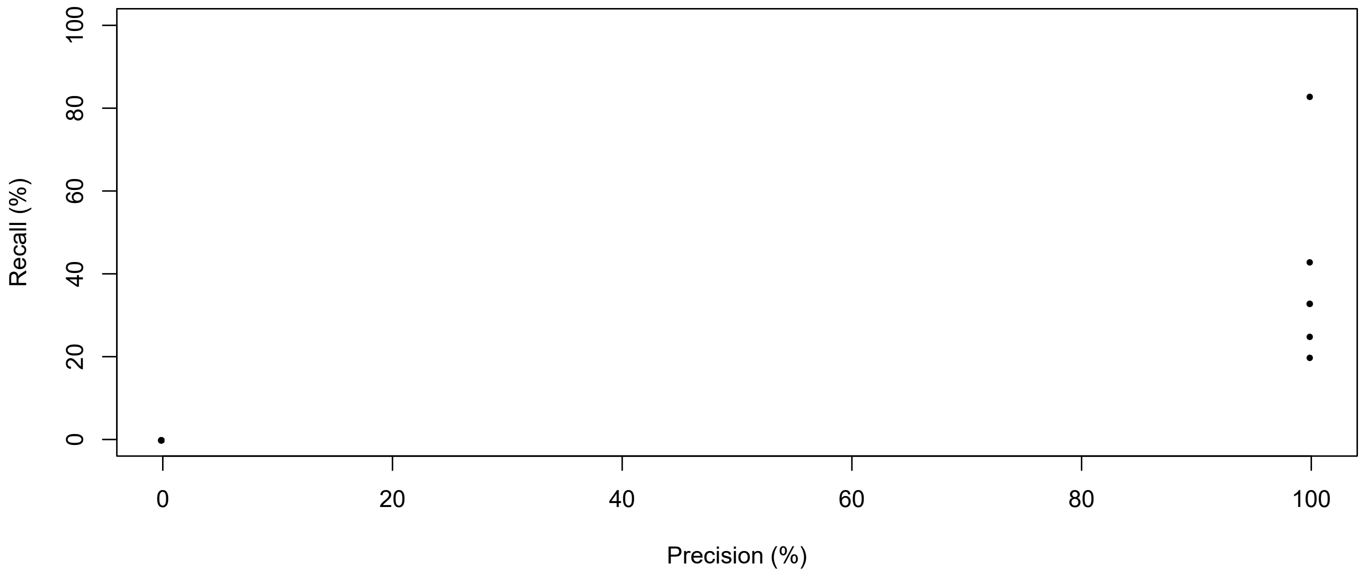


FIGURE A8 Mean Precision and Recall for the CAF Case Study for SDL

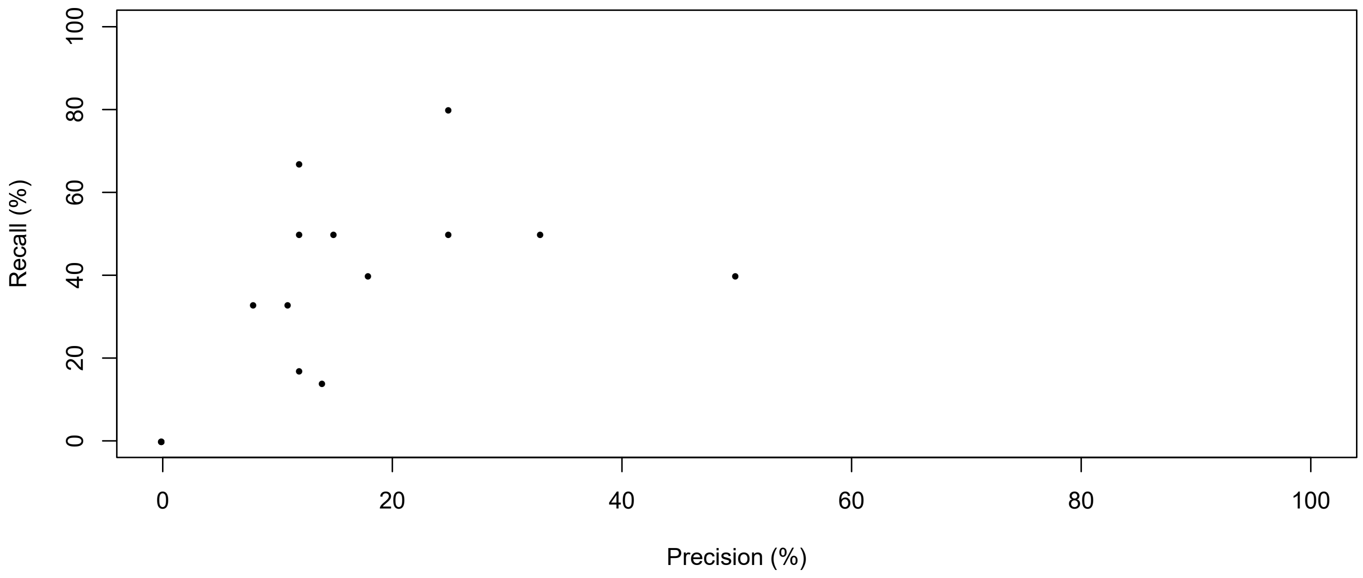


FIGURE A9 Mean Precision and Recall for the CAF Case Study for UML

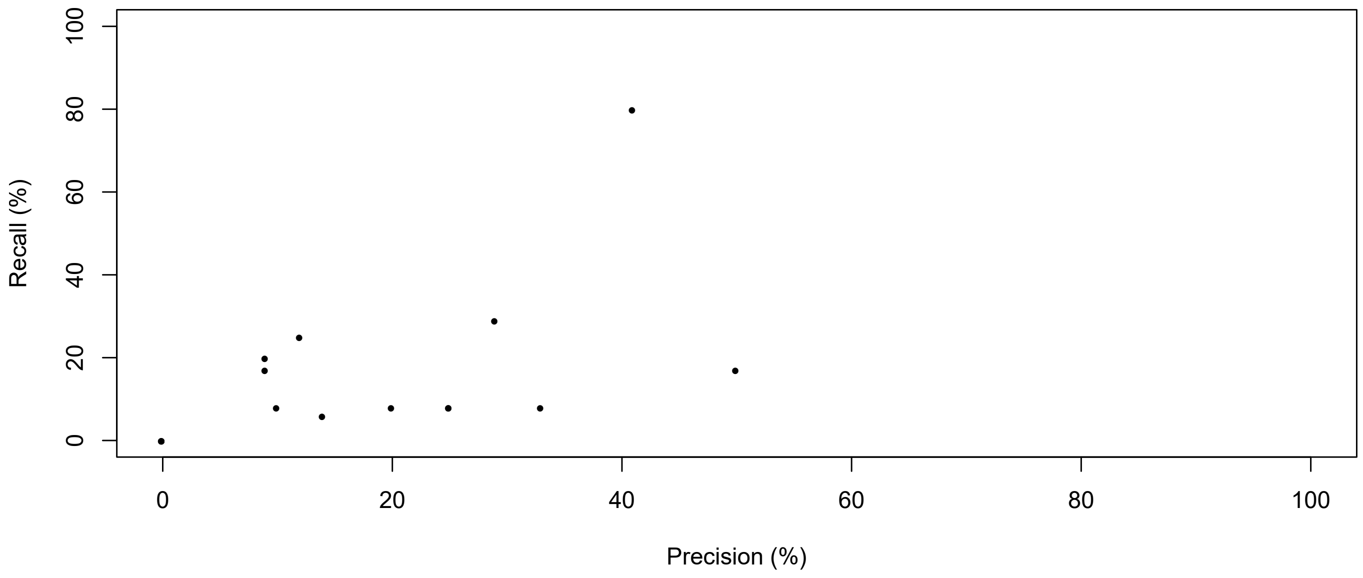


FIGURE A10 Mean Precision and Recall for the CAF Case Study for xADL

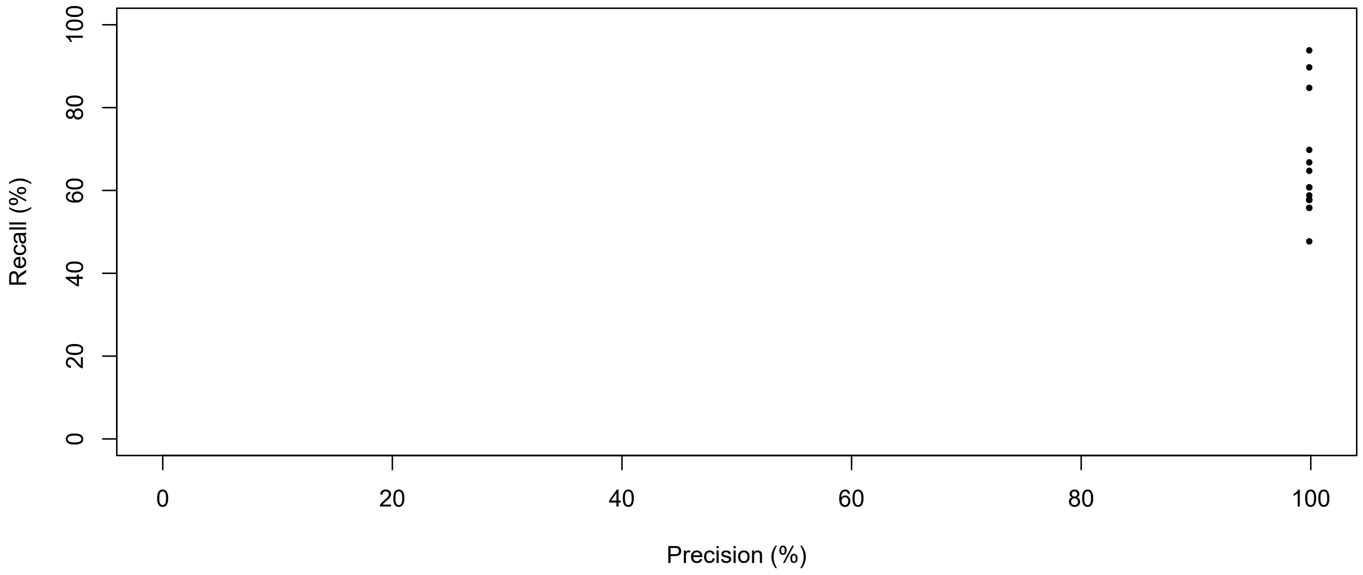


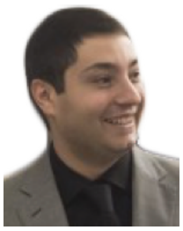
FIGURE A11 Mean Precision and Recall for the CAF Case Study for *Pantograph*

AUTHOR BIOGRAPHY

Manuel Ballarin. He is Researcher with the SVIT Research Group, Universidad San Jorge, Zaragoza, Spain. He received a Ph.D. degree in computer science from Universitat Politécnica de Valéncia (UPV). His current research interests include software product lines, model-driven development and software architectures. He publishes her research results and participates in high-level international software engineering conferences and journals, such as the International Conference on Model-Driven Engineering Languages and Systems (MODELS), and Information & Software Technology (IST) journal.

Lorena Arcega. She is Tenure Track Professor with the SVIT Research Group, Universidad San Jorge, Zaragoza, Spain. She received a Ph.D. degree in computer science from the University of Oslo, Oslo, Norway. Her current research interests include models at runtime, software maintenance and evolution, and variability modeling. She publishes her research results and participates in high-level international software engineering conferences and journals, such as the International Conference on Model-Driven Engineering Languages and Systems (MODELS), and Software and System Modeling (SoSyM) journal.

Vicente Pelechano. He is a full professor of software engineering at Universitat Politécnica de Valéncia (UPV). He received a Ph.D. in computer science from Universitat Politécnica de Valéncia. His research interests are Model-Driven Software Development, Autonomous Computing and Self-Adaptation, Service Engineering, Mobile and Ubiquitous Computing, Software Product Lines, Human–Computer Interaction, and Business Process Modeling. He has more than 190 research papers in renowned indexed journals (Multimedia Tools and Applications, JSS, IST, Transactions on the Web, etc.) and international conferences (ER, CAiSE, ICWE, Models, SPLC, etc.).



Carlos Cetina. He is Associate Professor with San Jorge University and the Head of the SVIT Research Group. He received a Ph.D in computer science from the Polytechnic University of Valencia. His research focuses on software product lines and model-driven development. His research results have reshaped software development in world-leading industries from heterogeneous domains ranging from induction hob firmware to train control and management systems. More information about his background can be found at his website: http://carloscetina.com