SN Computer Science (2025) 6:314
https://doi.org/10.1007/s42979-025-03843-3

ORIGINAL RESEARCH

An Empirical Investigation of Requirements Engineering and Testing Utilizing EARS Notation in PLC Programs

Mikael Ebrahimi Salari[1] · Eduard Paul Enoiu[1] · Wasif Afzal[1] · Cristina Seceleanu[1]

Received: 21 November 2023 / Accepted: 27 February 2025

Abstract

Regulatory standards for engineering safety-critical systems often demand both traceable requirements and specification-based testing, during development. Requirements are often written in natural language, yet for specification purposes, this may be supplemented by formal or semi-formal descriptions, to increase clarity. However, the choice of notation of the latter is often constrained by the designers' training, skills, and preferences. The Easy Approach to Requirements Syntax (EARS) addresses the inherent imprecision of natural language requirements concerning potential ambiguity and lack of accuracy. This paper investigates requirements specification using EARS, and specification-based testing of embedded software written in the IEC 61131-3 language, a programming standard for developing programmable logic controllers (PLC). Further, we study, utilizing an experiment, how human participants translate natural language requirements into EARS and how they use the latter to test PLC software. We report our observations during the experiments, including the type of EARS patterns that participants use to structure natural language requirements and challenges during the specification phase, and present the results of testing based on EARS-formalized requirements in real-world industrial settings.

Keywords EARS · Requirement engineering · PLC · Testing

Introduction

PLCs are used in engineering embedded safety-critical software (e.g., in the railway and automation control domains) [1]. The engineering of such systems commonly demands certification according to safety standards [2] that impose specific constraints on requirements engineering, implementation-based, and specification-based testing. Several studies [3–6] have examined how to generate test input data to achieve high implementation coverage for domain-specific PLC systems.

✉ Mikael Ebrahimi Salari
mikael.salari@mdu.se

Eduard Paul Enoiu
eduard.enoiu@mdu.se

Wasif Afzal
wasif.afzal@mdu.se

Cristina Seceleanu
cristina.seceleanu@mdu.se

[1] Department of Innovation, Design, and Engineering, Mälardalen University, Universitetsplan 1, 72220 Västerås, Västmanland, Sweden

However, since requirements are often expressed in natural language, using them as such to create test cases, and also keep requirements and test cases aligned, is a difficult task. While such an alignment requires extensive domain knowledge, a systematic process for requirements engineering—including their translation into a semi-formal, non-ambiguous form—combined with testing would facilitate linking requirements to tests. Generally, in industry, such translation is most often carried out manually, so manual processes are used to model requirements by using structured notations, and automatically create a set of tests that systematically exercises the specification when fed to the system under test [7]. Given that there is little evidence on the extent to which humans can effectively model requirements using semi-formal notations, and how the modeling impacts the development and testing of reliable systems, in this paper, we investigate the implications of applying structured requirements specification and test generation based on the impact of modeling on development and testing of PLC systems. In this context, we study how practitioners write requirements using the *Easy Approach to Requirements Syntax (EARS)* [8], a simple notation for specifying textual requirements in a structured and unambiguous manner.

Published online: 24 March 2025

We evaluate the EARS-based requirement modeling by involving human subjects. Ten individuals take part as subjects in an experiment. The subjects are given three requirements specified in natural language and are asked to rewrite them manually, using the EARS notation.

This work builds upon our previous work [9], and it extends it by a rigorous investigation of the applicability and efficiency of EARS-based testing of industrial PLC programs, including a comparison of EARS-based testing of PLC programs with testing the same programs manually. The results of our study show that humans create pattern-based requirements using semi-formal notations easily, with completeness being the most common issue when rewriting and using such requirements for testing. Additionally, we find that test generation and execution using the EARS requirements for PLC systems is a promising approach that applies to real-world industrial settings. Our results highlight the need for more research into how different requirement specifications and test design techniques for PLC software can influence the efficiency and effectiveness of requirements engineering and requirements-based testing for this type of software.

Preliminaries

Programmable Logic Controllers (PLCs)

PLCs are the most used logic controllers in today's automation industry [10]. PLCs are being widely used in different industrial applications such as supervisory systems in nuclear and power plants. Programming a PLC device is usually done via one or a combination of different programming languages that are proposed in the IEC 61131-3 standard [11], that is, *Function Block Diagram* (FBD), *Structured Text* (ST), *Ladder Diagram* (LD), *Sequential Function Chart* (SFC), and *Continuous Function Chart* (CFC). Among all programming languages for PLC, FBD, and ST are our main focus in this study for two reasons. First, these two languages have gained remarkable popularity in industry, during the last couple of years [12]. Second, the industrial case study that is provided to us for this study is a supervisory PLC program developed in ST and FBD. ST is a text-based programming language with a similar syntax to high-level programming languages such as C, whereas, FBD is a visual programming language that is easy to use due to its graphical interface. In PLC programs, a Program Organization Unit (POU) is a fundamental building block that consists of individual software components such as programs, function blocks, and functions. POUs enable modular and reusable code, allowing for efficient organization and management of complex control systems within a PLC. PLC programs are commonly developed in an Integrated Development Environment (IDE) and are executed cyclically. Based on the provided concept in IEC 61131-3 standard, each cycle loop of a PLC program execution consists of 3 main stages, that is, read, execute, write [11]. The first stage reads all available inputs and stores them in the memory, whereas the second stage (execute) carries out the computation tasks without interruption. The final stage (write) updates the output values based on the completed computations of the previous stage.

CODESYS Development Environment

Developing a PLC program and simulating its behavior needs to be done in an IDE. Several different PLC IDEs have been developed by different vendors so far. One of the most popular IDEs in the market is CODESYS,[1] which was initially developed by CODESYS Group in 1994. CODESYS is a manufacturer-independent IDE that has matured by releasing numerous updates and the latest version at this moment is V3.5 SP18. Among all available PLC IDEs in the market, We have chosen CODESYS as our preferred IDE for several reasons. Firstly, CODESYS is very popular among practitioners and has almost full compatibility with the IEC61131-3 standard and supports all proposed standard programming languages of this standard [12]. Secondly, CODESYS is free for personal use and is equipped with good support by releasing different versions. Last but not least, CODESYS can execute Python scripts directly inside the IDE, and it is also equipped with numerous automation add-ons, such as test automation tools.

The official CODESYS test automation tool is called CODESYS Test Manager. The CODESYS Test Manager tool is a powerful platform designed to streamline the testing process for PLC programs. It offers a user-friendly interface that simplifies test case creation, execution, and analysis, significantly enhancing efficiency and accuracy in PLC testing workflows. One of the key benefits of the Test Manager is its set of features, including predefined *Test Actions*, which automate various testing tasks and leverage the capabilities of the CODESYS IDE. Test Actions contain various functionalities, such as simulating inputs and outputs, monitoring variables, triggering events, and analyzing program behavior. For example, an "Assert" Test Action can be used to verify that a specific condition holds true during program execution, while a "Delay" Test Action can introduce time delays between test steps. These Test Actions allow testers to simulate real-world scenarios, validate program behavior, and generate detailed test reports, ultimately facilitating faster and more reliable PLC testing processes. an example of a Test Action can be observed in Fig. 1.

[1] https://www.codesys.com/.

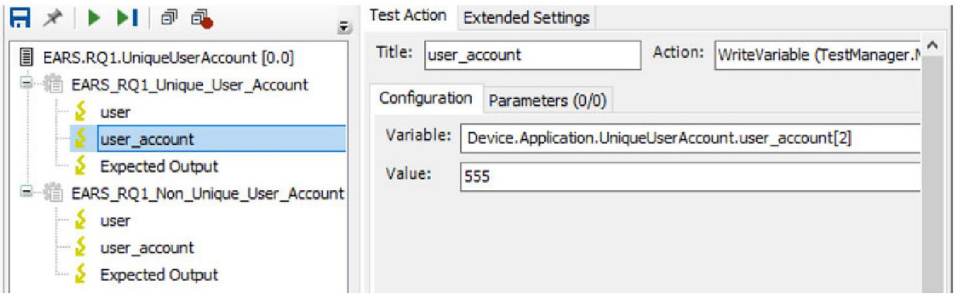

Fig. 1 The generated test cases for PRG1 based on the EARS syntax for RI1 via the built-in *Test Actions* of CODESYS Test Manager tool

EARS Semi-structured Requirement Engineering Syntax

Writing the stakeholder requirements in unconstrained Natural Language (NL) is not accurate and can raise critical problems in lower levels of system development [8]. Aiming at mitigating the ambiguity problems and increasing the accuracy in the process of requirements engineering, some practitioners argue for using other textual and non-textual notations [8]. Using non-textual notations demands translation of the original requirement, which can be faulty sometimes. Training overhead is another drawback of proposing a new type of notation. EARS is a semi-structured requirements engineering syntax transforming all-natural language requirements into one of the proposed five generic requirements syntax simple templates. It was initially proposed by Alistair et al. in 2009 [8]. EARS provides a syntax for transforming all-natural language requirements in one of the proposed five generic requirements syntax simple templates. The aforementioned five simple templates of EARS are ubiquitous requirements, event-driven requirements, unwanted behaviors, state-driven requirements, and optional features. Moreover, EARS supports writing complex requirements using a combination of considered conditional keywords, including *Where*, *While*, and *When*.

Experimental Design

In this section, we report the description of the performed experiment, including the details of the instruction material and the artefacts used.

Research Questions

The main goal of this study is to investigate the process of requirements creation when constraining the use of NL. The EARS modeling notation has been adopted by other organizations in different sectors and countries, so it is a realistic model for requirements engineering and test creation. Since these are intellectual activities in which humans allocate a variety of cognitive resources (such as attention and effort) that one needs to use when confronted with challenges as they perform such tasks, our first step is understanding how human practitioners write such requirements and how these can be used for test creation.

The main goal of this study is to investigate the applicability of the EARS semi-structured requirement engineering syntax in the context of PLC programs. Aiming to achieve this goal, we formulated the following research questions.

- RQ1: How are the EARS semi-structured requirement engineering syntax and test creation applied in the context of PLC programs?
- RQ2: What EARS patterns are used during the writing of requirements?
- RQ3: What challenges are perceived during the specification of requirements and test creation using EARS?
- RQ4: How well do PLC test cases created from EARS requirements compare to test cases created by industrial engineers for PLC programs in industry?

Experimental Setup Overview

Aiming to achieve this study's goal, we conduct a controlled experiment that asks the participants to write three given requirements using EARS syntax. The participants can choose their preferred EARS syntax template based on their interpretation of the requirements. The *subjects* of this experiment are a group of ten individuals as follows: four experienced engineers at a large automation company in Sweden and Spain and six researchers and managers from different universities and research institutions across Europe. All researchers involved in this experiment are postgraduate students with substantial expertise in the security and safety of cyber-physical systems.

Object Selection

The objects of study are chosen manually on the following criteria:

- The requirements should be specified using unambiguous, traceable natural language, for an engineer to be able to write executable tests.
- The requirements should represent different real testing scenarios in various areas where the IEC 61131-3 standard is used.
- The requirements should be simple to understand without any domain knowledge.
- The resulting test cases should be executed in the CODESYS environment.

We investigated the industrial libraries a large-scale company provides, focusing on developing and manufacturing control systems. We identified three candidate requirements matching our criteria, shown in Table 1. We then assess the relative difficulty of the identified requirements by manually writing and creating tests.

Operationalization of Constructs

Requirements templates. In this experiment, we investigate the effect of using the EARS approach for requirements engineering and test creation. The proposed generic requirements syntax of the EARS patterns we used in this experiment are structured as follows:

<optional preconditions> <optional trigger> the <system name> shall <system response>

This simple syntax template forces the requirement engineer to emphasize preconditions, triggers, and system responses in their developed requirements. In EARS syntax, preconditions and triggers are both optional, and the order of the used clauses is very important. The following briefly describes each template of EARS.

Ubiquitous Requirements (U)

A ubiquitous requirement is a type of requirement that is not bound to any preconditions or triggers and is always enabled in the system. The generic structure of this template is as follows:

The <system name> shall <system response>

Table 1 The natural language requirements used during the experiment

Requirement ID	Requirement text
RI1	User account should be uniquely identified to a user
RI2	The software shall warn the user of malware detection
RI3	Only authorized devices are allowed to connect to the ICS network

Event-Driven Requirements (ED)

The event-driven requirement is used only when an event is identified in the system. This type of requirement uses the *When* keyword. The generic structure of this template is as follows:

WHEN <optional preconditions> <trigger> the <system name> shall <system response>

Unwanted Behaviors (UB)

Requirements that are related to unwanted behaviors are defined using a structure extracted from event-driven requirements. *Unwanted behavior* refers to covering all possible situations that are not desirable and are usually a big source of omissions in preliminary requirements. The reserved keywords for this type of requirement in EARS are *If* and *Then*. The generic structure of this template is as follows:

IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>

It should be noted that there is a difference between ED and UB patterns; ED focuses on expected behaviors, while UB is looking into undesirable scenarios. More precisely, the first pattern should be used to cover positive system behavior and the second on negative scenarios or fault conditions.

State-Driven Requirements (SD)

The State-driven requirement is only active if the system is in a specific state. The reserved keyword for defining State-driven requirements in EARS is *While*. The generic structure of this template is as follows:

WHILE <in a specific state> the <system name> shall <system response>

Optional Features (OF)

The *Optional feature* requirement is designed to be used when the author wants to include a specific feature in the system. The keyword *Where* is considered for defining this type of feature in EARS. The generic structure of this template is as follows:

WHERE <feature is included> the <system name> shall <system response>

Process Challenges. We are interested in two types of challenges encountered during using EARS templates and their use for testing: challenges faced during the specification of requirements and problems when designing test cases for PLC systems. We performed thematic analysis [13] for qualitative data analysis to extract the main themes as reflected by the input given by each participant.

Instrumentation

One session was organized for the sake of the experiment. After introducing the EARS semi-formal requirement syntax to the subjects of this study via a short presentation, the subjects were given the task of using the three requirements and rewriting these in EARS (to the extent they consider sufficient based on the given specifications). They were instructed to read the specifications, create requirements from the EARS templates, and perform tasks without verbalizing their thoughts out loud. The subjects were not grouped, and the documents needed for this experiment were provided digitally and in written form. Before starting the session, a short tutorial of approximately 10 min on EARS syntax was provided to the subjects to avoid further problems with the subjects' unfamiliarity with the concepts used. The tutorial included screencasts demonstrating EARS requirements. Detailed information about the problem and instructions were provided in the experiment session.

Data Collection Procedure

As part of the instructions, subjects submitted their solutions in the form of a textual record documenting their work. Data from this experiment session was then used for quantitative and qualitative analysis.

Experiment Conduct

Once the experiment design was defined, the requirements for executing the experiment were in place. The session was held for one hour and preceded by a lesson on EARS notation. The requirements specification and testing process used during the conduct of this experiment corresponds to the methodology in Fig. 2. The first step corresponds to the transformation of the requirement specified initially in Natural Language (NL) into an EARS requirement using the EARS syntax (Step 1 in Fig. 2).

In the next step, we use the resulting requirement to generate test cases covering the specified behavior (Step 2 in Fig. 2). The final steps in this methodology are to execute these test cases (Step 3 in Fig. 2) and to compare the actual behavior with the expected result to monitor whether the program works as expected (Step 4 in Fig. 2).

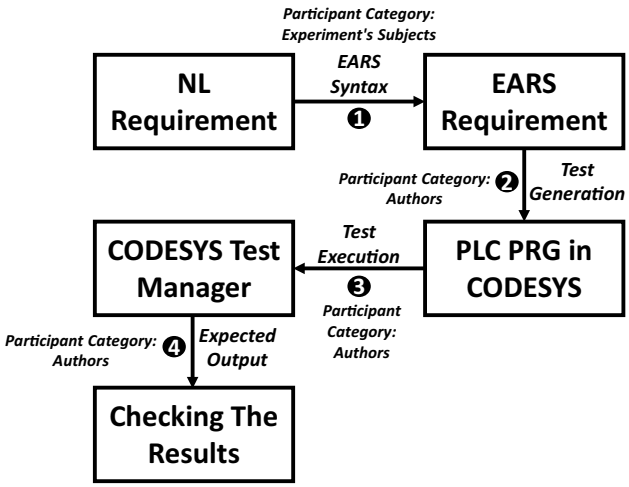

Fig. 2 An overview of the proposed EARS-based requirement specification and PLC testing methodology used in this experiment as well as the participant category of each step

In total, *ten individuals* participated in our experiment. Before starting the experiment, the participants were informed that their work would be used for experimental purposes. The participants had the option of not participating in the experiment and not allowing their data to be used this way.

The subjects worked individually during the experiment; we briefly interacted with the participants to ensure that everybody understood the involved notations without getting involved in the writing of the solution. All subjects used the provided documents and their machines. The experiment was fixed to one hour. To complete the assignment, the subjects were given the same time to work on writing these requirements according to the given instructions. For collecting data, we provided a template to enforce the usage of the same reporting interface. By having a common template for reporting, we eased the data collection and analysis process.

To finish the assignment, we required the participants to provide the produced results as soon as they finished writing their responses. During the experiment, the subjects do not directly communicate with others to avoid introducing bias. After each individual finished their assignment, a complete solution was saved containing the answers for each solution. In addition, we separated the data provided by the participants from their names. As shown in Fig. 2, the participants in Step 1 (marked in green) are the subjects of this experiment, including ten individuals, including industrial engineers and researchers. In contrast, Steps 2–4 represent the case study performed by the authors of this manuscript, as indicated in Fig. 2 by the blue markings.

Table 2 Number of participants using each EARS template for the corresponding requirement (RI1, RI2, RI3) in the experiment

RI1	RI2	RI3	Requirement ID/EARS template
10	1	1	Ubiquitous (U)
0	5	4	Event-driven (ED)
1	5	6	Unwanted behaviors (UB)
0	0	3	State-driven (SD)
0	0	0	Optional features (OF)

Table 3 Results of the requirements writing in terms of the templates used by each participant for each requirement. EARS template types are shown using their specific acronyms as stated in Sect. 3.4 and Table 2

RI1	RI2	RI3	Participant ID
U, UB	U, UB, ED	U, SD, ED	P1
U	ED	UB	P2
U	ED	UB	P3
U	UB	SD	P4
U	ED	UB	P5
U	ED	UB	P6
U	SD	UB	P7
U	UB	ED, UB, SD	P8
U	UB	ED	P9
U	UB	ED	P10

Experiment Analysis

This section provides an analysis of the data collected in this experiment. In analyzing the qualitative data, we followed the guidelines on qualitative analysis procedures provided by Braun and Clarke [13]. For each requirement, each subject in our study provided a set of EARS expressions. These expressions were used to conduct the experimental analysis and testing. For each set of tests produced, we provide evidence for their generation and execution in CODESYS. These metrics form the basis for our analysis toward answering the research questions.

Requirement Engineering Results

For each requirement, we have collected data about the type of EARS template used by each participant, the approaches, and the challenges participants experienced during requirement representation using the EARS notation. The results are shown in Tables 2, 3, and 4.

Participants strictly adhered to one or multiple EARS templates. It seems that the ubiquitous template has been used by all participants to model requirement RI1 and just in one case when representing requirements RI2 and RI3 (as shown in Table 2). The values in Table 2 represent the number of participants who selected each EARS template for the specified requirement. Participants explained that the "shall" statement is clearly indicated and should be used to describe the required behavior. However, a participant decided to use the unwanted behavior template for RI1 to indicate prohibited behavior in a form that can be used for testing.

The event-driven and unwanted behavior templates have been used by participants to represent requirement RI2, while some participants used the state-driven pattern (as shown in Table 3). Participants chose to do this since they drafted requirements in several increments. Firstly, they considered how the system typically behaves (also called sunny-day behavior). For some participants using EARS, this results in requirements in the state-driven and event-driven patterns. Secondly, some participants decided to specify what the system must do in response to the unwanted behavior, which produced requirements in the unwanted behavior pattern.

Table 4 Results showing the main themes identified related to approaches and challenges encountered during the translation process

Main themes	Theme descriptions
Requirements are not complete and clear enough for EARS translation	When starting with the translation, requirements in NL are not complete enough to decide precisely which EARS template to use
Using single or multiple EARS templates is not clear enough, especially when using these for testing	There is a need, when using these patterns for testing, to use multiple and separate templates for each requirement to cover both positive and negative cases arising
The system perspective is not easily identifiable from the requirements	It is difficult to decide which perspective to use when translating the EARS requirement (e.g., system, subsystem level)
The optional feature template is not applicable for the selected requirements	Even if the Option requirement is used for systems that include a particular element and variants, this modeling form was not used during requirement transformation using the EARS notation since the participants did not need to handle system or product variation

The results in Table 2 indicate that the Ubiquitous requirements are straightforward, encompassing general system behaviors that are consistently true. The participants perceived these as simple to write and understand. Many participants found these templates intuitive because they do not require complex conditions or states, making them easier to validate and test. Based on the extracted text and a general understanding of EARS templates, users prefer certain templates due to their simplicity, alignment with real-world scenarios, ability to handle the behavior explicitly, and flexibility in specifying state-dependent and optional behaviors. The observed preferences reflect the templates' alignment with participants' experiences and the specific needs of the tested systems. We have not conducted a detailed investigation into the reasons why certain templates are preferred over others; this observation is solely based on the results of our experiment.

In addition, the thematic analysis of the notes taken by participants when performing these steps in requirement representation resulted in several main themes related to approaches and challenges experienced during the translation process. Several participants mentioned that the initial NL requirements are not complete and clear such that these can be used directly for testing. One participant mentioned the following: *"What happens if the device is not authorized, missing failure models, startup/default/safe state...?"*. This resulted in issues when starting with the translation process, especially when deciding which templates to use. Several participants had issues in deciding when to use single or multiple EARS templates to cover both positive and negative behaviors that need to be tested. One participant stated the following: *"We could possibly use event-driven type requirement. At the same time, it is unwanted we could use, this one is quite complicated"*. Some participants preferred the use of the "shall not" form, which has been observed by some participants as having an impact on the test case created since only a set of test cases involving the unwanted behavior would need to be created to show satisfaction with the requirement. Another observation relates to the use of an optional feature template, which for the given requirements was not used by any of the participants since there was no need to specify any product variation or specific features.

Transformation of EARS Requirements and PLC Testing

To evaluate the applicability of using EARS semi-structured syntax when creating test cases for PLC programs, we used three programs that implement the behavior stated in the three provided natural language requirements used in this experiment. All these three PLC programs are developed in CODESYS IDE using the ST programming language, whether by the authors or derived from the existing industrial cases. In this paper, we refer to these programs as *PRG1*, *PRG2*, and *PRG3*.

We used the concretization steps of the EARS expressions as stated by Flemstrom et al. [14]. This happens by mapping the system response, condition, and events to the actual implementation in PLC. This contains information on the implementation elements of a system and its interfaces. An engineer needs to consider this information and identify the signals given and their characteristics. In this way, we define a set of signals related to the feature under test. In these cases, the next step for the selected requirements would be to design test cases to show that the requirement has been met.

The process of transforming system requirements into EARS requirements for PLC testing is a structured approach, as shown in Fig. 3. It begins with the requirement analysis phase, where system requirements and design documentation are used. Next, the information collected is formalized into abstract EARS requirements in the abstract EARS requirement construction phase. These requirements use logical names for entities, making communication with requirements engineers easier, though they are not directly evaluable at this stage. The implementation analysis phase then involves mapping these abstract entities to actual implementation elements by analyzing design documents. This includes identifying the signals or events corresponding to the abstract entities defined earlier. The process then proceeds to the EARS requirement concretization phase, where the abstract EARS requirements are transformed into concrete counterparts using actual signals and events from the system. This results in a set of concrete EARS requirements ready for the creation and execution of tests, as shown in Table 5.

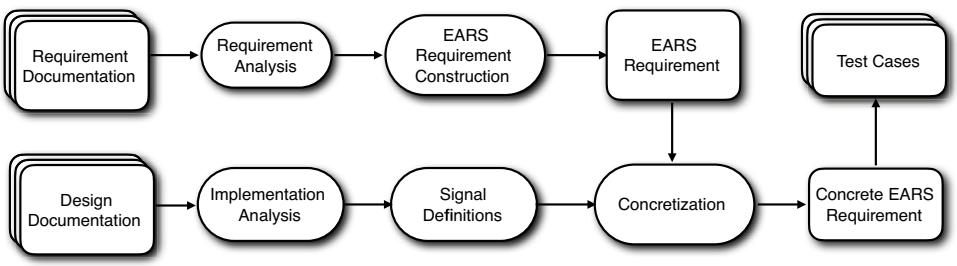

Fig. 3 Process overview for transforming requirements to EARS for PLC testing

Table 5 EARS requirements examples obtained from the experiment and the resulting concretized EARS requirements and example test cases

Req.	EARS requirements	Concretized EARS requirements	Example test cases
RI1	The < user account system > shall < identify the user > If < the user is not identified > then < user account system > shall < alert >	if < uniqueID=FALSE> then < UniqueUserAccount> shall < Result_Unique=FALSE >	**Description:** Verify system alerts when a user is not identified. **Steps:** 1. Attempt to identify a user with invalid credentials. 2. Check uniqueID. 3. Verify alert is generated
RI2	When < malware is detected> the < system > shall < warn the user >	When < NormalActivity ≠ MaliciousActivity> the < MalwareDetection> shall < MalwareDetected=TRUE >	**Description:** Verify system warns user when malware is detected. **Steps:** 1. Simulate malicious activity. 2. Check activity status. 3. Verify warning is issued
RI3	When < the device is authorised > the < system> shall < grant access to the device>	When < found=TRUE > the < SearchID > shall < ConnectionAllowed=TRUE >	**Description:** Verify system grants access to an authorized device. **Steps:** 1. Simulate an authorized device. 2. Check device status. 3. Verify access is granted

To create test cases for requirement RI1, which involves user identification and alerting, we set up a scenario where the system attempts to identify a user with invalid credentials, ensuring the *uniqueID* is FALSE. We then verify that the system generates an alert by checking if *UniqueUserAccount* results in *ResultUnique=FALSE*. This involves preparing the system to identify users, attempting identification with invalid data, and checking for the appropriate alert response.

To create a test case for RI2, we simulate malicious activity in the system, ensuring that *NormalActivity* does not equal *MaliciousActivity*. We then verify that the *MalwareDetection* component sets *MalwareDetected* to TRUE and that the system issues a warning to the user. This involves running the system to monitor for malware, simulating malicious activity, and checking the detection and the warning response.

For requirement RI3, to create a test case, we simulate the presence of an authorized device, ensuring that it is found (*found=TRUE*). We then verify that the *SearchID* component sets *ConnectionAllowed* to TRUE and that the system grants access to the device. This involves configuring the system to authorize devices, simulating the authorization of a device, and checking that access is granted correctly.

After generating the EARS-based test cases for each program, we execute these automatically using the CODESYS test automation framework named CODESYS Test Manager.[2] The final step in this methodology is to manually compare the actual output with the expected output to observe whether the program works as expected.

To clarify the key roles in each part of the experiment, Fig. 2 illustrates that we first conduct a controlled experiment (Step 1 in Fig. 2). Subsequently, using the resulting EARS requirements, we perform a case study and transform these requirements into test cases (Steps 2, 3, and 4 in Fig. 2).

Test Results of *PRG1*

PRG1 is the PLC program we considered for testing the RI1 requirement in the PLC environment. This program is using the values of the *user account* and *user* lists. Then it checks for unique IDs and returns an indication of whether each user account is uniquely identified to a user or not. A snippet of the PRG1 PLC program is shown in Listing 1.

To design and execute the required test cases to test the RI1 Requirement in PRG1, we use the transformed requirement from the NL requirement shown in Table 5.

[2] https://store.codesys.com/en/codesys-test-manager.html.

```
PROGRAM UniqueUserAccount
VAR
   user : ARRAY[1..10] OF WSTRING;
   user_account : ARRAY[1..10] OF DINT;
   i,j : INT;
   K : INT;
   UniqueID : BOOL; (*Non-Unique ID counter*)
   Result_Unique: BOOL := FALSE;
END_VAR
```

Listing 1 A listing showing part of the PRG1 PLC interface program written in the ST language in CODESYS IDE corresponding to the evaluation of the RI1 requirement

Based on the EARS requirement, we use two test cases to cover the identification of the user and the case when the user is not identified. Each test case includes the following three *Test Actions*: two *WriteVariable Test Actions* to alter the *user* and *user account* inputs and one *CompareVariable Test Action* that compares the actual output with the expected one. The generated test cases for PRG1 used to test the adherence of the program to RI1 requirements are shown in Fig. 1.

After designing the required test cases, we execute them automatically on PRG1 to investigate the adherence of the mentioned PLC program to the RI1 requirement. As can be observed in Fig. 4, all test cases have been executed in 0.3 s. All executed test cases have successfully passed on the PRG1 program.

Test Results of *PRG2*

The PLC program we use for executing the generated test cases for *RI2* in Table 1 is named PRG2. This program is shown as a black-box malware detection system in the PLC

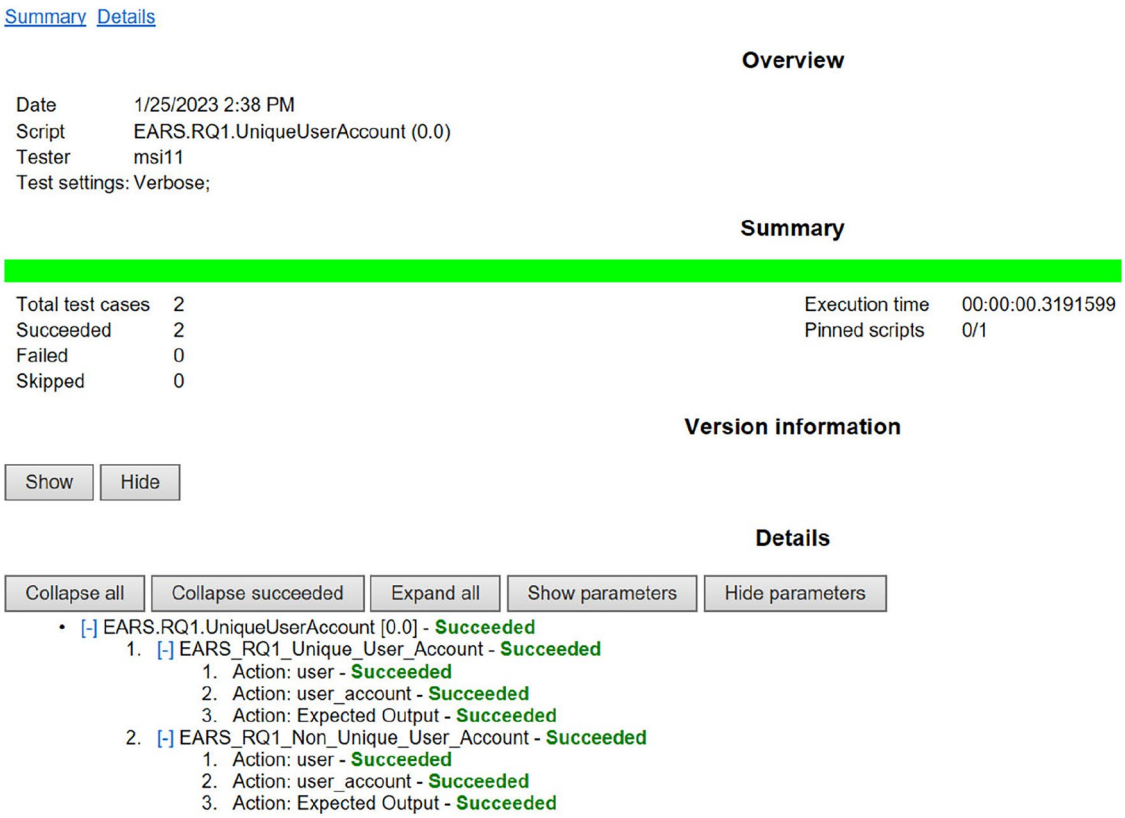

Fig. 4 Test execution results for PRG1 PLC program based on the EARS-based generated test cases for RI1

environment that can be used to investigate the context of RI2. PRG2 consists of the following interfaces: two input signals named *MaliciousActivity* and *NormalActivity* as well as one output signal named *MalwareDetected*. When *MaliciousActivity* and *NormalActivity* signals have divergent information, the Malware Detection system is triggered, and the value of the *MalwareDetected* signal becomes True. An interface snippet of PRG2 is shown in Listing 2.

```
PROGRAM MalwareDetection
VAR
    MalwareDetected: BOOL;
    MaliciousActivity: BOOL;
    NormalActivity: BOOL;
END_VAR
```

Listing 2 A listing showing part of the PRG2 PLC interface program written in the STlanguage in CODESYS IDE corresponding to the evaluation of the RI2 requirement

Considering the results of the experiment, we use the resulting EARS *Event-driven requirement* pattern as the most suited type of template for transforming the requirement from NL to EARS in the form shown in Table 5.

Based on the developed EARS requirement for RI2 requirement, we generate two test cases for PRG2. Each test case consists of two *Test Actions* (*MaliciousActivity* and *NormalActivity*) that alter the value of the inputs, as well as one *Test Action* (*Expected Output* that compares the actual behavior with the expected one. The first test case checks if a (*Malware is Detected*) while the second test case checks if a (*Malware is Not Detected*)

The generated test cases for PRG2 based on the RI2 requirement are then automatically executed using CODESYS Test Manager in 1.71 s. All developed test cases have successfully passed.

Test Results of *PRG3*

PRG3 is the PLC program used to execute the generated test cases for *RI3* in Table 1 (*"Only authorized devices are allowed to connect into the ICS network"*. This program consists of the following units: (1) a database of authorized device IDs, which is implemented using an array of IDs, (2) an input signal corresponding to the device ID that needs to be authorized, and (3) a boolean output signal (i.e., *found*) which returns True in the case of the authorized device being allowed to connect given the ID is known. We show a snapshot of this PLC program in Listing 3.

```
PROGRAM SearchID
VAR
  id_to_find : INT := 111;
  found : BOOL;
  array_of_ids : ARRAY[0..9] OF INT :=
[000,111,222,333,444,555,666,777,888,999];
  i : INT;
END_VAR
```

Listing 3 A listing showing part of the PRG3 PLC program written in the ST language in CODESYS IDE corresponding to the evaluation of the RI3 requirement

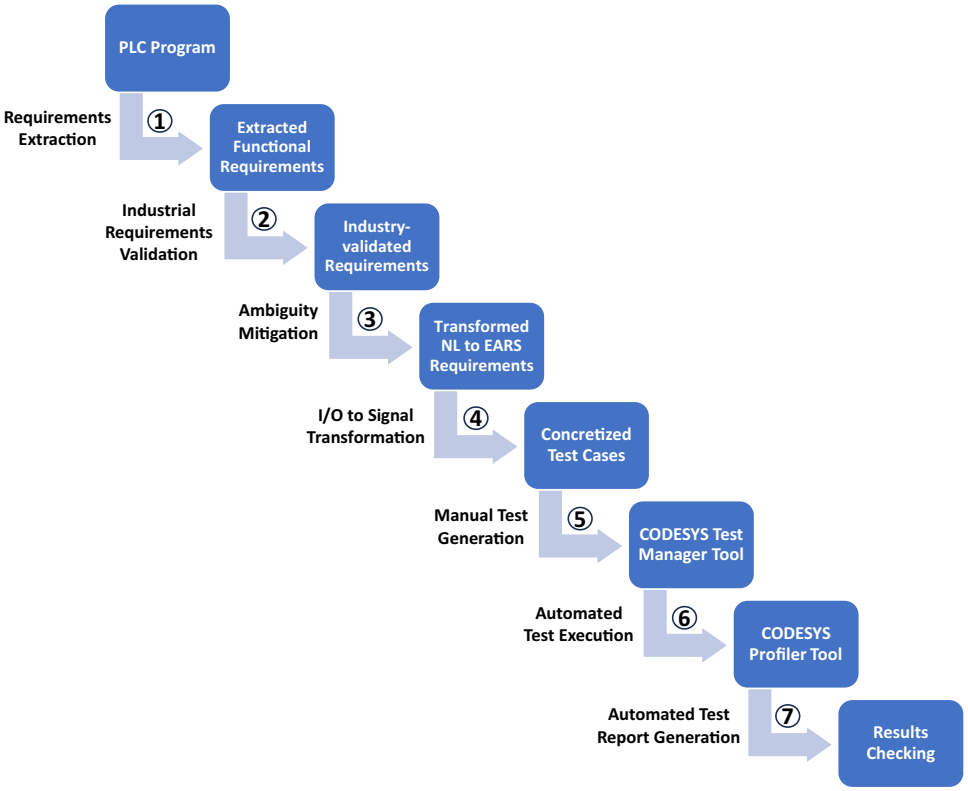

Fig. 5 The proposed EARS-based testing method for real-world industrial PLC testing

As discussed in Sect. 5.1, different individuals transformed the NL requirement into the EARS requirement in different forms. We use the most common form developed by the participants to transform RI3 to an EARS *Event-Driven* syntax pattern in the following form shown in Table 5.

Based on the aforementioned EARS requirement for RI3, we developed 2 test cases for *Successful Authorization* and *Unsuccessful Authorization*. Each developed test case consists of two *Test Actions*, including the provision of a *new Input ID* and *Comparing the actual output with the expected output*. The generated test cases have been automatically executed on PRG3 using CODESYS Test Manager in 1.14 s. Both test cases have successfully passed after being executed on the PRG3 PLC program.

EARS-Based Testing in Real-world Industrial Settings

To expand our investigation of the applicability and efficiency of PLC testing using EARS patterns in real-world industrial settings (RQ4), in this section, we extend our evaluation by including a real-world PLC program that is being used in the context of crane supervision by a large automation company in Sweden. To be more specific, we compare the EARS requirement-based test cases with real-world test scripts that are being used by the industry for PLC testing. We believe the conduction of this comparison can reveal hidden facts about the applicability and efficiency of using EARS-based testing versus the current real-world PLC testing in the industry.

Methodology for EARS-Based Testing in Real-World Industrial Settings

The methodology we propose for using EARS-based testing in real-world industrial settings consists of seven steps and is shown in Fig. 5. The first step is to extract the functional requirements from the real-world PLC program (step 1 in Fig. 5). The purpose and process of functional requirements extraction in the context of this study were necessary for the experiment, as we lacked requirements at this level. The second step is to have a team of industrial PLC engineers evaluate the validity of the functional requirements (step 2 in Fig. 5). The next step is to transform the NL requirements into EARS requirements to mitigate the potential ambiguity and increase the clarity of the extracted requirements for the tester (step 3 in Fig. 5). As the next step, the EARS requirements need to be concretized to facilitate the test generation by converting the Inputs/Outputs (I/O) into signals (step 4 in Fig. 5). After having the concretized test cases, it is time to manually generate test cases via the pre-defined *Test Actions* inside the CODESYS Test Manager tool (step 5 in Fig. 5).

The next step is to automatically execute the test cases on the PLC program using the CODESYS Test Manager tool (step 6 in Fig. 5). The final step in this methodology is to enhance the generated test report of CODESYS Test Manager by measuring the code coverage automatically using the CODESYS Profiler tool and checking the results (step 7 in Fig. 5).

Real-World Industrial PLC Program

In this section, we start by introducing the included real-world PLC program in this study by defining its purpose and functionality. Then, we analyze the industrial test script of this PLC program and compare it to our proposed EARS-based testing approach.

The included real-world PLC program in this work is called *CraneNumberCheck* and is shown in Listing 4. This simple but critical PLC program is a POU within a more complex PLC program that supervises the cooperative functionality of large industrial cranes in a port. This POU must check whether the crane numbers match or mismatch to generate a flag based on this information in the crane's supervision system. As can be observed in Listing 4, this PLC program is developed in ST language and is composed of two input variables which represent the crane numbers and are called *Crane_1* and *Crane_2* (Lines 2–6 in upper box in Listing 4). Moreover, this PLC program consists of two output variables called *Matched_Crane_No* and *out_Safe_Crane_No* (Lines 7–10). The first one checks if the crane numbers match and are not equal to `EmptyWord`, whereas the latter checks if the crane number is safe and if crane numbers mismatch, this word is set to an empty word.

```
1  (* Initialization Part of the PLC Program *)
2  FUNCTION_BLOCK CraneNumberCheck
3  VAR_INPUT
4      Crane_1: WORD; (* First crane number *)
5      Crane_2: WORD; (* Second crane number *)
6  END_VAR
7  VAR_OUTPUT
8      Matched_Crane_No: BOOL; (* Crane numbers match and are not
       an empty word *)
9      out_Safe_Crane_No: WORD; (* A safe crane number. If crane
       numbers mismatch, this word is set to an empty word *)
10 END_VAR
11 VAR
12     EmptyWord: WORD;
13 END_VAR
14
15 (* Code Part of the PLC Program *)
16 Matched_Crane_No :=
17     (Crane_1 = Crane_2)
18     AND (Crane_1 <> EmptyWord);
19
20 IF Matched_Crane_No THEN
21     out_Safe_Crane_No := Crane_1;
22 ELSE
23     out_Safe_Crane_No := EmptyWord;
24 END_IF
```

Listing 4 A listing showing the CraneNumberCheck PLC program as a real-world industrial case study in the context of port crane supervision program

As can be seen in the rest of the code in Listing 4, the functional logic of the PLC program consists of two main parts and works as follows. In the first part, the *Matched_Crane_No* is set to True if the crane numbers are equal and *Crane_1* is not an empty *Word* (Lines 16–18 in the bottom box of Listing 4). The second part of the PLC program's logic checks whether the crane numbers are matched. In case of success, the program returns the safe crane number; otherwise, it returns an empty *Word* (Lines 20–24 in the bottom box in Listing 4).

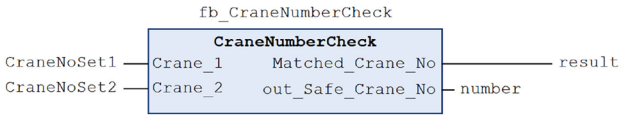

Fig. 6 A snapshot showing the function block instantiation of *CraneNumberCheck* POU inside the main PLC program to prepare it for testing/

```
1  (* Initialization, puls starting block *)
2  _RTRIG1(CLK:= in_Start, Q=> );
3  (* Enable testing case by case *)
4  IF (in_Start OR _RTRIG1.Q) AND in_StartC THEN
5      in_StartC := FALSE;
6  END_IF
7  (* Initialization of variables for test control *)
8  IF (_RTRIG1.Q) OR (in_StartC AND (step >=
       EnumValueSTest_TestDone)) THEN
9      FOR i := 1 TO 7 BY 1 DO
10         out_Status.Case_Ok[i] := FALSE;
11     END_FOR
12     out_ActCase := 0;
13     out_All_Ok := FALSE;
14     out_Status.Done := FALSE;
15     step := 1;
16 END_IF
17 (* Test process *)
18 IF in_Start OR in_StartC THEN
19     (* Delay between test steps and pulse generator *)
20     (* Setting of timer, every case will be long minimally 250ms
        *)
21     _TON1(IN:= step = stepOld, PT:= timerTime, Q=>, ET=> );
22
23     (* Default state needed for all test cases --- Area for
       default setting *)
24     craneNoSet1 := 0;
25     craneNoSet2 := 0;
26
27     (* Test procedure *)
28     CASE step OF
29         1:  (* Set for case 1 *)
30             (* All inputs are same as in area for default
       setting *)
31             out_ActCase := 1;
32
33         2:  (* Set for case 2 *)
34             (* All inputs are same as in area for default
       setting except input -> craneNoSet1 *)
35             craneNoSet1 := 3;
36             out_ActCase := 2;
```

Listing 5 A listing showing part of the real-world test script for testing the CraneNumberCheck PLC program in the current industry

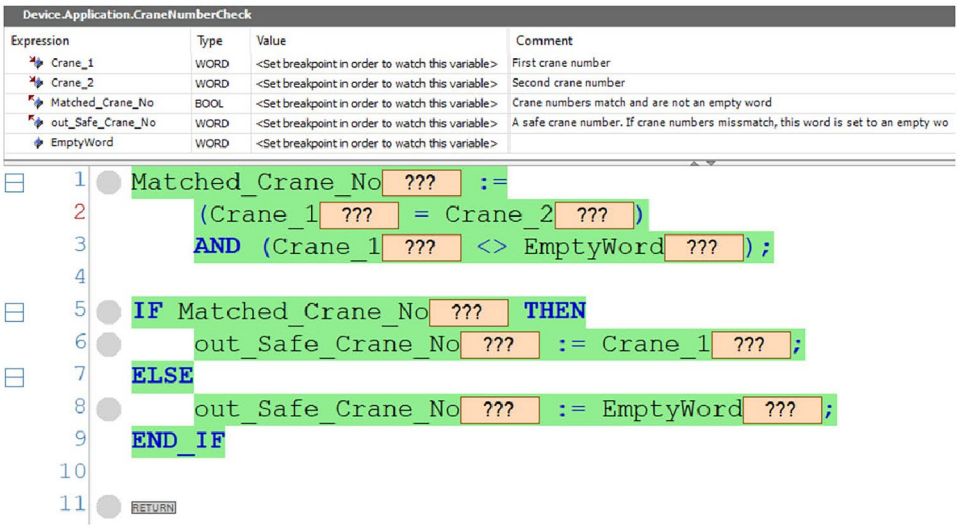

Fig. 7 A snapshot showing the CODESYS Profiler report on gathered full coverage for *CraneNumberCheck* PLC program using the proposed EARS-based method (refer to Fig. 5)

Testing of the Real-world Industrial PLC Program Under Test

The current testing process of the *CraneNumberCheck* PLC program in industry is handled by manually developing a counterpart testing *POU* in ST language. Part of the real-world industrial test script used for testing this PLC program is shown in Listing 5. As can be observed in Listing 5, the industrial test script consists of several main steps.

It starts with the initialization of a puls starting block as a trigger for starting the testing process, followed by an *IF* condition for enabling the test cases one by one (Lines 1–6 in Listing 5). The next step is initializing variables for test control (Lines 7–16 in Listing 5). After setting up the initialization, the next step is to define the main testing process, which includes setting up a delay between test steps and the pulse generator, followed by setting up a timer function that simulates the cyclic execution behavior of PLC programs (Lines 18–26 in Listing 5). The rest of the testing process of *CraneNumberCheck* PLC program consists of unit test cases that define inputs and expected output. The authors are responsible for artificially creating the PLC system and conducting the corresponding EARS-based testing. The EARS requirements have been independently derived from the functional requirements extracted during the controlled experiment. The subsequent transformation into test cases is a concretization step, involving the mapping of abstract EARS requirements to specific testable elements (e.g., signals and outputs) in the PLC system.

Results of EARS-Based Testing of a Real-World Industrial PLC Program

In this section, we use the proposed EARS-based testing methodology (refer to Fig. 5) for testing the *CraneNumberCheck* PLC program as a real-world industrial case study. The first step is to extract the functional NL requirements of the PLC program (Step 1 in Fig. 5). The second step is to have a team of industrial PLC engineers evaluate the validity of the extracted functional requirements (step 2 in Fig. 5). Table 6 shows the extracted functional unit-level NL requirements for this PLC program, which were all validated by a team of experienced PLC engineers at ABB Ports and Marine in Sweden. This table also includes the used EARS pattern and the EARS version of each requirement which is described as step 3 of our proposed methodology in Fig. 5.

As can be observed in RQ2 and RQ3 rows of Table 6, the extracted functional requirements in NL can sometimes become complicated and hard to follow for developers while their EARS versions in the "Requirement in EARS" column are modularized and much easier to comprehend for the PLC program developer/testers. Moreover, we can observe that one complicated NL requirement can break into several smaller EARS requirements, increasing the readability of the requirements.

After having the functional requirements in the EARS syntax, we take the next step of our methodology, which is to concretize the EARS requirements to generate unit test cases (step 4 in Fig. 5). The procedure for concretizing the EARS requirements for PLC testing is simple and works as follows. Each I/O inside the requirement is transformed into a signal, which can facilitate the test generation process as the next step. The concretized version of each EARS requirement for *CraneNumberCheck* PLC program is shown in Table 7.

The next step in testing *CraneNumberCheck* PLC program based on the proposed testing approach is to generate test cases based on the concretized EARS requirements in the previous step (Step 5 in Fig. 5). To do this, first, we need to instantiate *CraneNumberCheck* PLC program as a function block inside the main PLC program. A snippet of the function block that we instantiated for *CraneNumberCheck* PLC program can

Table 6 Industry-validated unit-level functional requirements for *CraneNumberCheck* PLC program (refer to Listing 4) in both NL and EARS versions

No	Functional requirements	EARS pattern	Requirement in EARS
RQ1	"The function block shall accept two crane numbers (Crane_1 and Crane_2) as input parameters. (Input Requirements)"	Ubiquitous requirement (U)	The system shall accept two crane numbers (Crane_1 and Crane_2) as input parameters
RQ2	"The function block shall provide two output variables: Matched_Crane_No: This variable indicates whether the crane numbers match and are not an empty word. out_Safe_Crane_No: If the crane numbers match, this variable shall store a safe crane number. If the crane numbers do not match, it shall be set to an empty word. (Output Requirements)"	Unwanted behaviors (UB)/ State-driven requirements (SD)	UB: IF the crane numbers match and are not an empty word, THEN the system shall set Matched_Crane_No to true. SD: WHERE the crane numbers match, the system shall set out_Safe_Crane_No to a safe crane number. SD: IF the crane numbers do not match, THEN the system shall set out_Safe_Crane_No to an empty word
RQ3	"The function block shall implement the following logic: Matched_Crane_No shall be true if Crane_1 is equal to Crane_2 and both are not empty words. If Matched_Crane_No is true, out_Safe_Crane_No shall be set to Crane_1. If Matched_Crane_No is false, out_Safe_Crane_No shall be set to an empty word. (Logic)"	State-driven requirement (SD)	WHERE the function block is active, the system shall: - Set Matched_Crane_No to true IF Crane_1 is equal to Crane_2 and both are not empty words. - Set Matched_Crane_No to false IF Crane_1 is not equal to Crane_2 OR either of them is an empty word. - IF Matched_Crane_No is true, THEN set out_Safe_Crane_No to Crane_1. - IF Matched_Crane_No is false, THEN set out_Safe_Crane_No to an empty word
RQ4	"The function block expects that an empty word is a valid condition for not matching crane numbers. (Constraints)"	Ubiquitous requirement (U)	The system shall consider an empty word as a valid condition for not matching crane numbers

be observed in Fig. 6. As the next step, we used CODESYS Test Manager to design the test cases using the pre-defined *Test Actions* of this tool. After automatic execution of test cases on the *CraneNumberCheck* PLC program and using the CODESYS Profiler tool for measuring code coverage (step 6,7 in Fig. 5), we gathered the following results. All the designed test cases with a timeout budget of 1 s have been successfully passed within 12 s on the PLC program under test. Moreover, the automatic test execution based on the proposed EARS-based PLC testing method for real-world industrial PLC programs achieved 100% code coverage on *CraneNumberCheck* PLC program based on the CODESYS Profiler report. A snippet of gathered full code coverage after testing the *CraneNumberCheck* PLC program is shown in Fig. 7. As shown in Fig. 7, all the covered code branches after executing EARS-based test cases have been marked green. The gathered results promise an acceptable level of applicability and efficiency of the proposed EARS-based testing method in the context of PLC programming. However, more investigation by applying this method to more complicated PLC programs needs to be done to validate the generalizability of this claim.

EARS-Based Testing vs Manual PLC Testing in Industry

Comparing the overall current manual testing process of *CraneNumberCheck* PLC program in the industry versus the proposed EARS-based testing mechanism reveals several observations, including:

1. **Need for domain-specific knowledge.** One needs to have a good understanding of one of the IEC61131-3 programming languages to develop test cases for the PLC program in the current industrial approach. To be more precise, the current manual industrial testing method requires the tester to write a complete PLC test script in one of the IEC61131-3 programming languages, creating a parallel POU to the main POU being tested. Additionally, the tester must perform continuous testing of the PLC program by simulating its cyclic execution behavior using one of the IEC61131-3 languages. This approach demands significant attention to detail and expertise, as it involves accurately mimicking the PLC's execution cycles. Moreover, the manual tester needs a testing background and engineering experience to implement and connect all testing units properly. On the other hand, testing the PLC programs with the proposed mechanism using CODESYS Test Manager does not demand any deep knowledge of specific programming language and can be handled easily using *Test Actions*.
2. **Efficiency.** Considering a straightforward PLC program like *CraneNumberCheck*, which is composed of 25 Lines of Code (LOC), the corresponding real-world industrial test script requires 119 LOC, highlighting a

Table 7 The concretized requirements of *CraneNumberCheck* PLC program (refer to Fig. 4) which are generated based on the requirements in EARS syntax

Requirement in EARS	Concretized requirement
The system shall accept two crane numbers (Crane_1 and Crane_2) as input parameters	if < Crane_1 & Crane_2 Exist=FALSE>then < SystemAcceptance>shall < Acceptance=FALSE>
UB: IF the crane numbers match and are not an empty word, THEN the system shall set Matched_Crane_No to true. SD: WHERE the crane numbers match, the system shall set out_Safe_Crane_No to a safe crane number. SD: IF the crane numbers do not match, THEN the system shall set out_Safe_Crane_No to an empty word	UB: if < Crane_1 = Crane_2 & Crane_1 & Crane_2 ≠ Empty>then < Matched_Crane_No>shall < Matched_Crane_No=TRUE>SD: WHERE < Crane_1 = Crane_2>the < System>shall < out_Safe_Crane_No=SafeCraneNumber>SD: IF < Crane_1 ≠ Crane_2>THEN < System>shall < out_Safe_Crane_No=EmptyWord>
The function block shall implement the following logic: Matched_Crane_No shall be true if Crane_1 is equal to Crane_2 and both are not empty words. If Matched_Crane_No is true, out_Safe_Crane_No shall be set to Crane_1. If Matched_Crane_No is false, out_Safe_Crane_No shall be set to an empty word	WHERE the function block is active, the system shall: - Set Matched_Crane_No to true IF Crane_1 is equal to Crane_2 and both are not empty words. - Set Matched_Crane_No to false IF Crane_1 is not equal to Crane_2 OR either of them is an empty word. - IF Matched_Crane_No is true, THEN set out_Safe_Crane_No to Crane_1. - IF Matched_Crane_No is false, THEN set out_Safe_Crane_No to an empty word
The system shall consider an empty word as a valid condition for not matching crane numbers	if < Crane_1 ≠ Crane_2>then < SystemAcceptance>shall < Acceptance=Empty Word>

significant discrepancy in efficiency. In contrast, the proposed EARS-based testing approach requires only 26 *Test Actions*. A *Test Action* in CODESYS Test Manager is a predefined step or operation configured within test scripts to communicate with the test system. These actions manage tasks such as setting communication parameters, executing tests, and generating reports. Compared to traditional lines of code, *Test Actions* simplifies the test creation. They encapsulate specific operations in a user-friendly manner, reducing complexity by providing clear, modular steps.

3. **Manual overhead and complexity.** The industry's current testing process for PLC programs is highly complex, with significant manual intervention. Specifically, many features already present in the CODESYS IDE, such as cyclic execution, delay, test control process, and test start trigger, are being recreated manually. This redundancy exacerbates the complexity, especially with more intricate PLC programs. Conversely, the proposed EARS-based testing approach simplifies this by requiring the manual definition of only the inputs and expected outputs. All other features are readily accessible through the user-friendly GUI of the CODESYS Test Manager tool. Additionally, the availability of pre-defined *Test Actions* within the Test Manager tool enhances the use of CODESYS' features and automation capabilities for PLC testers. Migrating to the latest version of the CODESYS IDE is an ongoing process for many automation companies due to its powerful features. However, the utilization of its testing capabilities has not received sufficient attention from industry practitioners, primarily due to practical limitations and established preferences.
4. **Test Report.** The existing manual testing process in the industry offers testers limited information, providing only the outcomes of passed or failed test cases. In contrast, the proposed EARS-based testing approach utilizes both CODESYS Test Manager and CODESYS Profiler to provide a comprehensive test report. These include additional details like test execution time, coverage reports, outcomes of individual *Test Actions*, test verdicts, and more.
5. **Ambiguity and clarity of functional requirements.** After reviewing a limited set of requirements gathered from the industry, it became apparent that the current functional requirements are predominantly at the system level, lacking specificity for individual code branches. Additionally, the complexity of industrial testing processes relies heavily on the tester's expertise. In contrast, the proposed EARS-based approach reduces the vagueness of requirements and encompasses both unit and system-level testing, potentially leading to a more thorough testing procedure. Furthermore, this approach yields requirements and test cases that are straightforward and understandable, facilitating understanding among all stakeholders, including testers, managers, and clients.

Discussion, Imitations, Threats to Validity

This section briefly discusses the findings of this work and maps them to each related formulated research question. This follows by discussing the limitations of this study and threats to validity.

Research Questions Revisited

In this subsection, we briefly review the formulated research questions of this study and align them with the corresponding results obtained.

RQ1: "How are the EARS Semi-structured Requirement Engineering Syntax and Test Creation Applied in the Context of PLC Programs?"

Results-RQ1: This study assessed the use of EARS syntax for creating test cases for three PLC programs in ST language (PRG1, PRG2, PRG3) developed in CODESYS IDE. EARS-based test cases were generated and executed with the CODESYS Test Manager. PRG1, testing unique user IDs, passed all cases in 0.3 seconds. PRG2, a malware detection system, passed all cases in 1.71 seconds. PRG3, verifying authorized device IDs, passed all cases in 1.14 seconds. The tests were derived by translating natural language requirements into EARS syntax, forming the basis for specification-based testing to ensure the programs met their requirements. The results of this study indicate the applicability and efficiency of EARS notation in test generation for PLC programs.

RQ2: "What EARS patterns are used during the writing of requirements?"

Results-RQ2: The study analyzed the use of EARS templates by participants, focusing on their approaches and challenges in requirement representation, as shown in Tables 2, 3, and 4. Most participants used the ubiquitous template for RI1 and occasionally for RI2 and RI3. They also employed event-driven, state-driven, and unwanted behavior templates. Key challenges included unclear initial requirements, difficulty selecting appropriate templates, and deciding on single versus multiple templates for positive and negative behaviors. Participants highlighted issues with "shall not" statements impacting test cases and found no need for the optional feature template.

RQ3: "What challenges are perceived during the specification of requirements and test creation using EARS?"

Results-RQ3: The study addresses challenges encountered when using EARS notation for requirement representation and subsequent test case generation. Participants encountered difficulties stemming from unclear initial requirements, selecting appropriate EARS templates, deciding on single or multiple templates, and defining the system perspective. These complexities underscore the challenge of translating natural language requirements into structured formats like EARS. Ambiguities and incompleteness in the initial requirements hindered precise translation into EARS syntax, complicating test case derivation. Determining suitable test coverage for positive and negative behaviors based on EARS requirements added further complexity. These challenges emphasize the intricate nature of both requirement representation and test case generation using the EARS syntax, highlighting the importance of clarity and careful consideration throughout the process.

RQ4: "How well do PLC test cases created from EARS requirements compare to test cases created by industrial engineers for PLC programs in industry?"

Results-RQ4: Comparing manual PLC testing with EARS-based testing for the CraneNumberCheck program reveals significant advantages of the latter. The EARS-based approach uses straightforward Test Actions, reduces intricacy through automation, and provides comprehensive test specifications. This approach enhances the clarity of functional requirements, leading to more detailed testing. The results suggest that EARS-based testing can improve testing efficiency and effectiveness for PLC programs.

Limitations of the Study and Threats to Validity

External validity. All of our subjects are individuals who have limited experience with EARS. Furthermore, because these practitioners have experience in requirements engineering, we see no reason the use of professionals with deep knowledge of EARS in our study would yield a completely different result. Professionals with experience in EARS would intuitively write better requirements than the ones written by our subjects. Our study has focused on three relatively brief examples with reduced complexity, but these requirements represent relevant samples they would encounter in practice. We have used the CODESYS tool for automated test creation and execution. There are many tools for developing and executing tests, and these may give different results. Nevertheless, CODESYS is one of the most used development environments for PLCs, and its output in tests is similar to the output produced by other tools.

Internal validity. All subjects were assigned to experiment at the same time. This was dictated by how the experiment was organized, with a presentation followed by practical work. Subjects without sufficient knowledge of EARS may affect the final result. To avoid this problem, the session was structured to follow the corresponding EARS lesson. Another threat to internal validity could arise from using unclear objectives given to the subjects. To address this, we tested the material ourselves.

Constructs validity. Capturing the challenges of requirements engineering and testing is a difficult problem. We rely on human feedback by using a think-out-loud method that gives a rough measure of the challenges encountered. Moreover, a potential threat to validity is the authors' involvement in transforming requirements into test cases. While this ensured consistency when using a PLC development environment, it introduces potential biases, as the authors have been involved in the case study. To mitigate this, participants have performed the initial requirement formalization, independently.

Conclusion validity. The results of this study are based on an experiment using 10 participants and three requirements. For each requirement, all participants performed the study, which is a relatively small number of subjects. Nevertheless, this was sufficient to obtain various results showing an effect between the modeling of these different types of requirements. Another limitation of this experiment is that participants were not asked to formulate testable requirements using EARS patterns. However, our primary focus was not on their testability, as the translation step performed by participants did not involve direct concretization. This step was completed after the data collection phase.

Related Work

Requirement engineering (RE) for testing embedded systems plays a crucial role in ensuring the reliability and functionality of these systems, which are ubiquitous in various domains, including automotive, aerospace, and healthcare. Model-checking, a formal verification technique, has emerged as a promising approach for formally verifying embedded system requirements. This section explores recent advancements and insights in the intersection of requirement engineering for testing embedded systems and model-checking.

Mavin and Wilkinson [15] reflected on the ten years of EARS [8] and shared some lessons learned in their review paper. For example, they found that EARS users manage to author more useful draft requirements as they incrementally work to find the appropriate EARS pattern. They recommend that new engineers write several requirements and seek expert review with the application of EARS being more useful if one can apply the following activities: training, thinking, semantics, syntax, and review. In our study, we confirm some of these results even if we do not cover all of the activities stated.

Mavin et al. [16] report on the understanding of four experienced EARS practitioners and their reflections on their experiences of applying EARS in different projects and domains over six years. They report the following EARS-specific lessons learned: training should be short, use EARS with or without a tool, use coaching to embed learning, challenge the EARS Patterns, and question if the EARS clauses are necessary and sufficient.

Mäntylä et al. [17] performed a controlled experiment on test case development and requirement review and the effects of time pressure. They saw no statistically significant evidence that time pressure would lower effectiveness or negatively influence motivation, frustration, or performance.

Dalpiaz et al. [18] investigated the adequateness, completeness, and correctness of use cases and user stories for the manual creation of a static conceptual model. They performed a controlled experiment with 118 subjects, and their results show that user stories work better than use cases when creating conceptual models. Furthermore, user story repetitions and conciseness contribute to these results. However, as we aim with our study, more evidence needs to be provided regarding the aspects that must be considered when selecting and using a modeling and requirement notation.

Weninger et al. [19] report the results of a controlled experiment in which they compared two approaches for defining restricted use case requirements from multiple perspectives, including misuse, understandability, and restrictiveness. Their results indicate the usefulness of the restricted use case modeling approach.

The paper by Levi Lucio et al. [20] introduces the EARS-CTRL tool, an editor built on MPS, designed to aid in the crafting and analysis of EARS requirements for controllers. This editor inherently ensures well-formedness, offering a structured method and suggesting relevant terms from a glossary during the editing process. The paper discusses automated checks for the feasibility of requirements, utilizing a controller synthesis tool, and the generation of synchronous dataflow diagrams for verified requirements. While it recognizes the challenges in representing complex states, the research emphasizes the importance of closing the gap between natural language requirements articulated in EARS and formal specifications, thereby enhancing the automation of requirement analysis and synthesis within an industrial setting.

The embedded systems industry demands specialized RE methods to effectively manage the complexity of requirements specifications and ensure high-quality outcomes. However, the adoption of novel RE approaches by industry practitioners remains slow. To address this, Sikora et al. [21] conducted an industrial study aimed at gaining a comprehensive understanding of practitioners' needs in RE research and method development. Through qualitative interviews and quantitative questionnaires, they explored five key aspects of RE approaches: (i) the use of requirements models, (ii) support for high system complexity, (iii) quality assurance for requirements, (iv) the transition between RE and design, and (v) the integration of RE with safety engineering. They conducted their study with representatives from seven large companies across multiple branches and identified significant industry needs and constraints. Their key findings include the advocacy for increased use of models in RE, challenges surrounding the use of requirements models in legally binding documents, and the critical importance of method support for abstraction layers. Additionally, practitioners emphasized the need for solutions addressing requirements for quality assurance, particularly in consistency, traceability, and testability. While the automated transition from requirements to design was not deemed a prevalent need, the study highlighted the importance of integrating safety engineering concerns into RE approaches. Compared to our work, this work focuses on the embedded systems industry, using qualitative and quantitative methods to identify broad RE challenges such as model usage, system complexity support, and safety engineering integration. In contrast, our work targets PLC systems, conducting experiments with EARS notation to assess its effectiveness and flexibility in RE and testing. While Sikora et al. identify industry needs and constraints, our research demonstrates the practicality of EARS-based test cases and suggests future automation.

Iqbal et al. [22] underscore the key role of requirement validation within the automotive industry, where the accuracy of embedded systems' functionalities directly impacts product integrity. They identify a prevailing challenge: conventional validation methods often prove inefficient, resulting in heightened project failure rates. The authors propose an innovative model-based approach to tackle this issue, harnessing existing verification and validation frameworks. Central to their methodology is the integration of virtual prototyping, enabling early-stage error detection. Through meticulous case studies, the authors showcase the benefits of their approach, including heightened productivity, simplified development cycles, and improved product quality. The key differences between this study and our study are as follows. Their work focuses on the automotive industry, proposing a model-based validation approach that integrates virtual prototyping to improve early error detection, productivity, development cycles, and product quality. In contrast, our work targets PLC systems, using EARS notation to evaluate its effectiveness and flexibility in RE and testing. While Iqbal et al. aim to enhance validation efficiency in automotive embedded systems, our research demonstrates the practicality of EARS-based test cases and suggests future automation for PLC RE.

In summary, recent advancements in requirements engineering for testing embedded systems have witnessed the

integration of model-checking techniques to improve formal verification and test case generation processes. These developments underscore the importance of leveraging test automation methods to ensure embedded systems' reliability and safety.

Conclusions and Future Work

In this paper, we have conducted an experiment in requirements engineering and testing using EARS notation for PLC systems. In the requirements engineering part of our experiment, we found out that most participants preferred the EARS ubiquitous pattern for transforming the RI1 requirement from NL to the EARS syntax, whereas the unwanted behavior and event-driven patterns were the most popular types for RI2 and RI3 requirement transformations. It was observed that different individuals used different EARS patterns for transforming the same requirement based on their interpretation, which shows an acceptable level of flexibility in EARS syntax. In the testing part of our experiment, we assessed the use of EARS patterns for PLC testing in two phases. Initially, we executed EARS-based test cases on three PLC programs written in the ST language, which were developed based on the requirements included in our study. Subsequently, we introduced an EARS-based testing methodology to real-world industrial PLC programs. The results from these tests and the subsequent comparison with traditional PLC testing methods indicate that EARS-generated requirement-based test cases for PLC programs are effective and offer an accessible means for PLC testers to express test specifications.

In future work, we want to investigate the applicability of using EARS in PLC requirement engineering on other levels of testing and by including more PLC programs. Inspection of the impact of choosing different EARS templates for describing the requirements over the quality of the generated test cases can be another future direction of our work. Moreover, we want to automate our solution and generate test cases from the created EARS requirements based on existing functional and non-functional requirements.

Acknowledgements This work has received funding from the EU's H2020 research and innovation program under grant agreement No 957212, Vinnova through the SmartDelta project, and the MATISSE project, an EU-funded initiative under Horizon Europe GA no. 101056674.

Author Contributions Mikael Ebrahimi Salari is the main driver and contributor of the paper, while the rest of the authors contributed to the methodology and provided valuable feedback.

Funding Open access funding provided by Mälardalen University. EU's H2020 research and innovation program under grant agreement No 957212 and from Vinnova through the SmartDelta project.

Data Availibility Synthetic data and also data from industry. The programs and data used in this study include proprietary industrial data that cannot be shared publicly.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Research involving human and/or animals Not applicable.

Informed consent Informed consent was obtained from all individual participants included in the study.

References

1. Schwartz MD, Mulder J, Trent J, Atkins WD. control system devices: architectures and supply channels overview. In: Sandia Report SAND2010-5183. Sandia National Laboratories, Albuquerque, NM, 2010.
2. CENELEC: 50128: railway application–communications, signaling and processing systems-software for railway control and protection systems. In: Standard Report, 2001.
3. Enoiu EP, Čaušević A, Ostrand TJ, Weyuker EJ, Sundmark D, Pettersson P. Automated test generation using model checking: an industrial evaluation. Int J Softw Tools Technol Transf. 2014;18(3):335–53.
4. Wu Y-C, Fan C-F. Automatic test case generation for structural testing of function block diagrams. Inf Softw Technol. 2014;56(10):1100–12.
5. Jee E-S, Yoo J-Y, Cha S-D, Bae D-H. A data flow-based structural testing technique for FBD programs. Inf Softw Technol. 2009;51(7):1131–9.
6. Doganay K, Bohlin M, Sellin O. Search based testing of embedded systems implemented in IEC 61131-3: an industrial case study. in: international conference on software testing. Verification and Validation Workshops. Luxembourg: IEEE; 2013. p. 425–32.
7. Garousi V, Zhi J. A survey of software testing practices in Canada. J Syst Softw. 2013;86(5):1354–76.
8. Mavin A, Wilkinson P, Harwood A, Novak M. Easy Approach to requirements syntax (EARS). In: 2009 17th IEEE International Requirements Engineering Conference. IEEE; 2009. p. 317–22.
9. Salari ME, Enoiu EP, Afzal W, Seceleanu C. An experiment in requirements engineering and testing using EARS notation for

PLC systems. In: 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE; 2023. p. 10–7.
10. Auslander DM, Pawlowski C, Ridgely J. Reconciling programmable logic controllers (PLCs) with mechatronics control software. In: Proceeding of the 1996 IEEE International Conference on Control Applications. IEEE; 1996. p. 415–20.
11. Tiegelkamp M, John K-H. IEC 61131-3: programming industrial automation systems, vol. 166. Berlin, Heidelberg: Springer; 2010.
12. Hanssen DH. Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS. Chichester: Wiley; 2015.
13. Braun V, Clarke V. Thematic analysis. Washington, DC: American Psychological Association; 2012.
14. Daniel F, Enoiu E, Azal W, Sundmark D, Gustafsson T, Kobetski A. From natural language requirements to passive test cases using guarded assertions. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE; 2018. p. 470–81.
15. Mavin A, Wilkinson P. Ten years of EARS. IEEE Softw. 2019;36(5):10–4.
16. Mavin A, Wilkinson P, Gregory S, Uusitalo E. Listens learned (8 lessons learned applying EARS). In: 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE; 2016. p. 276–82.
17. Mäntylä MV, Petersen K, Lehtinen TO, Lassenius C. Time pressure: a controlled experiment of test case development and requirements review. In: Proceedings of the 36th International Conference on Software Engineering, p. 83–94, 2014.
18. Dalpiaz F, Sturm A. Conceptualizing requirements using user stories and use cases: a controlled experiment. In: International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer; 2020. p. 221–38.
19. Weninger M, Grünbacher P, Zhang H, Yue T, Ali S. Tool support for restricted use case specification: findings from a controlled experiment. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE; 2018. p. 21–30.
20. Lúcio L, Rahman S, Cheng C-H, Mavin A. Just formal enough? Automated analysis of EARS requirements. In: NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16–18, 2017, Proceedings 9. Springer; 2017. p. 427–34.
21. Sikora E, Tenbergen B, Pohl K. Industry needs and research directions in requirements engineering for embedded systems. Requir Eng. 2012;17:57–78.
22. Iqbal D, Abbas A, Ali M, Khan MUS, Nawaz R. Requirement validation for embedded systems in automotive industry through modeling. IEEE Access. 2020;8:8697–719.