The Journal of Systems and Software 212 (2024) 112028

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In practice

Custom static analysis to enhance insight into the usage of in-house libraries ☆

Piërre van de Laar [a,*], Rosilde Corvino [a], Arjan J. Mooij [a,d], Hans van Wezep [b], Raymond Rosmalen [c]

[a] TNO-ESI, Eindhoven, The Netherlands
[b] Philips, Best, The Netherlands
[c] ITEC, Nijmegen, The Netherlands
[d] Zürich University of Applied Sciences, Winterthur, Switzerland

ARTICLE INFO

Keywords:
Customizable static analysis
Knowledge extraction and retrieval
Reverse engineering
Program comprehension
Code regularity
Industrial case study

ABSTRACT

For software maintenance and evolution, insight into the codebase is crucial. One way to enhance insight is the application of static analysis to extract and visualize program-specific relations from the code itself, such as call graphs and inheritance trees. Yet, software often contains in-house libraries: unique, domain-specific libraries whose usage is typically scattered throughout the codebase. To provide sufficient insight into the usage of those libraries, the static analysis must be customized with domain-specific information.

In this paper, we propose a method to enhance insight into the usage of in-house libraries by producing custom overviews. Furthermore, we describe three exploratory case studies targeting industrial C++ and Ada codebases, in which the method was developed, evolved, and validated.

The method prescribes how to create custom overviews using static analysis iteratively, starting from a user-provided, initial specification of proper library usage using code patterns. As a safeguard, the method includes cross-checks to detect code fragments that deviate from proper library usage. Whenever such a deviating library usage is found, the code owners determine whether that deviating library usage should be added to the specification of proper library usage or the code fragment should be made compliant. The latter alternative makes both the codebase more regular and keeps the custom static analysis simpler. The method creates custom overviews that reveal opportunities to improve the usage of the in-house libraries, e.g., the removal of domain-specific redundant code which cannot be detected using generic tools, such as compilers and linters.

We observed that industrial codebases are regular enough to create custom overviews using static analysis in the three exploratory case studies. Furthermore, we observed that the cross-checks, which detect deviating library usage, ensure the validity and completeness of the custom overviews. We conclude that producing custom overviews for in-house libraries using the method is valuable and feasible.

1. Introduction

Several studies (Schröter et al., 2017; Xia et al., 2018) show that software engineers spend most of their time understanding source code, not only for maintenance and modernization but also for evolution and development. Software is often built around in-house libraries with various purposes, such as data exchange, logging, and company-specific functionality. Software engineers[1] would benefit from enhanced insight into the usage of those in-house libraries, whose usage is typically scattered throughout the codebase. In this paper, we address the research question: *How to enhance insight into the usage of in-house libraries?*

In-house libraries provide abstractions relevant to a specific domain. The domain concepts are exposed using a combination of programming language constructs, such as functions, classes, and constants. The usage of these programming constructs typically adheres to particular code patterns, either as required by the library's preconditions and protocols or due to the software engineers' explicit practices and implicit conventions. We aim for insight in terms of these domain concepts instead of the programming language concepts.

☆ Editor: Marcos Kalinowski.

* Corresponding author.
E-mail address: pierre.vandelaar@tno.nl (P. van de Laar).

[1] The type of an engineer is not exclusive. A software engineer must understand software yet might also be a mechanical engineer, a systems engineer, or a hardware engineer.

https://doi.org/10.1016/j.jss.2024.112028
Received 17 April 2023; Received in revised form 1 February 2024; Accepted 14 March 2024
Available online 19 March 2024

For example, consider a time measurement library that introduces the stopwatch domain concept and implements it using the following programming language constructs: a stopwatch class with member functions to start and stop the time measurement. Imagine a codebase that uses these member functions as follows:

- the stop function is only called for started stopwatches, as required by the library; and
- a stopped stopwatch is, in practice, never started again, although this is supported by the library.

Given these usage patterns, an overview of the time measurement library could show for each stopwatch the location, or absence thereof, where it is constructed, started, and stopped. This overview enhances insight in the codebase by exposing the relation between the code fragments associated with each stopwatch. Furthermore, this overview enables domain-specific improvements to the codebase: started-but-never-stopped stopwatches correspond to incomplete time measurements and all code related to these stopwatches is domain-specific redundant code which can be removed.

Manual analysis of large codebases is laborious and error-prone, so we aim for automation, with the following requirements:

- automation must be able to process all relevant code of the application, and
- automation must be customizable to the specific analysis in the specific domain.

We considered the following options for automation:

- *Regular expressions:* Analysis using regular expressions is supported in most text editors, integrated development environments, and programming languages. Regular expressions are however text-based and thus sensitive to comments and white spaces. Moreover, regular expressions cannot handle programming language structures like recursively nested parentheses.
- *Cross-reference databases:* Many compilers, with the appropriate flag (often `xref`), and dedicated tools, like cxref,[2] GNATxref,[3] and the dependency graph extractor of Renaissance for C/C++ and IDL (Dams et al., 2021) and of Renaissance-Ada (van de Laar and Mooij, 2022), can summarize a codebase in a database containing all symbols and references. Analysis can exploit such a cross-reference database. However, only few analyses need just the information captured in a cross-reference database.
- *Linters:* Linters analyze a codebase for general issues, such as bugs, style violations, and non-portable constructs. Some linters, like GNATcheck[4] and SonarQube,[5] support custom rules. However, many analyses need more customization than is provided by linters, as, e.g., reported by Mendonça et al. (2018).
- *Parsers:* Many parsers, such as Eclipse CDT,[6] Clang,[7] ASIS,[8] and Libadalang,[9] provide an Application Programming Interface (API) to enable custom static analysis. These parsers typically also provide basic functionality for syntactic and semantic analysis. However, software engineers often experience a steep learning curve, especially when the parser's abstract syntax tree is exposed.

Since regular expressions are unable to process all programs, and since cross-reference databases and linters cannot be customized sufficiently, we have selected parsers to enhance insight into the usage of in-house libraries.

[2] https://www.gedanken.org.uk/software/cxref.
[3] https://www.adacore.com/gnatpro/toolsuite/utilities.
[4] https://www.adacore.com/static-analysis/gnatcheck.
[5] https://www.sonarsource.com/products/sonarqube.
[6] https://github.com/eclipse-cdt.
[7] https://clang.llvm.org.
[8] https://www.adacore.com/asis.
[9] https://adaco.re/libadalang.

Software "understanding involves dealing with specific problems that require program and task-specialized solutions" (Reiss, 2005). So, for in-house libraries, one should not expect off-the-shelf analysis tools to be immediately effective. For this reason, we distinguish in-house libraries from widely-used off-the-shelf libraries, for which some static analysis specialists may already have developed some dedicated analyses. Customization is needed to find sweet spots in information abstraction: domain-specific insight without overwhelming details.

Jbara and Feitelson (2014) "suggest that code regularity – where the same structures are repeated time after time – may significantly reduce complexity, because once one figures out the basic repeated element it is easier to understand additional instances". This does not only apply to humans but also to tools. As Bessey et al. (2010) state "tools want expected". The benefits of code regularity are not limited to static analysis. Both Wlodarski et al. (2019) and Mooij et al. (2020) argue that automated code transformation also benefits from improving code regularity first, and Ossendrijver et al. (2022) states that the quality of the transformation "is dependent on how idiomatic and consistent the codebase is".

Irregularities can easily be introduced in a codebase as industrial codebases are developed and maintained for decades by an evolving multi-disciplinary team of engineers: Engineers have personal programming habits based on earlier experience and education; best practices in software engineering advance over time; and programming languages, standards, and libraries regularly get updated. Irregular usages of an in-house library deviate from the most-frequent usages of that library's API. However, these irregularities typically do not violate the usage constraints of the library's API and thus are typically not API misuses (Amann et al., 2019). The need for custom static analysis raises the question whether the codebases are regular enough to capture all code fragments relevant for that analysis with a limited amount of code patterns. At the start of our case studies it was not clear how to determine this regularity, so we have decided to observe this regularity experimentally.

In contrast to our earlier work (Klusener et al., 2018; Mooij et al., 2020) on code transformation, in this paper we focus on code analysis. In particular, we propose a method to create custom overviews that enhance insight into the usage of in-house libraries. Furthermore, we describe three exploratory case studies involving industrial C++ and Ada codebases, in which the method was developed, evolved, and validated using the industry-as-laboratory approach (Potts, 1993). These case studies were executed within public–private partnership research projects. These projects were proposed by our industrial partners, as being in need of an innovative solution to enhance insight into the usage of their in-house libraries. In these case studies, we address two additional research questions specifically for code regularity:

- Are these industrial codebases sufficiently regular regarding the usage of in-house libraries so that custom static analysis is effective and efficient?
- Can code fragments that deviate from expected code patterns be detected effectively and efficiently to ensure the validity and completeness of the custom overviews?

The remainder of this paper is organized as follows: Section 2 presents the method to analyze the usage of in-house libraries. Sections 3, 4, and 5 describe an industrial case study of a blackboard library, a logging library, and an inspection library, respectively. The threats to validity are addressed in Section 6, and in Section 7 we discuss related work. We end with discussions, conclusions, and future work in Sections 8, 9, and 10.

2. Method

The method is summarized in Fig. 1 and has 3 phases: *develop*, *apply*, and *act*. We first describe these phases and briefly introduce the process

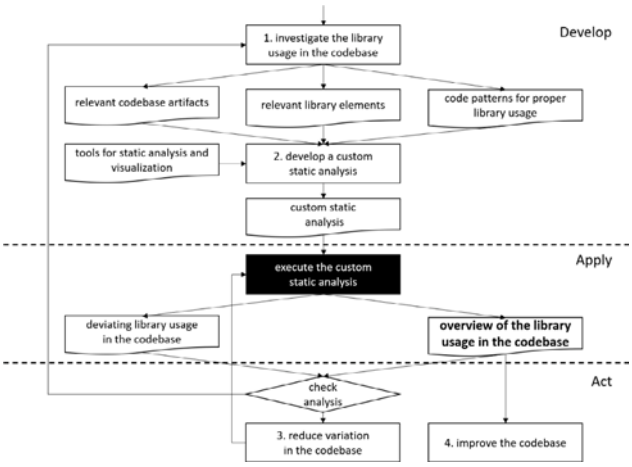

Fig. 1. Schematic summary of the method.

steps that they contain, then in the subsections we elaborate in more detail on these steps.

In the *develop* phase, a custom static analysis is developed in two steps. Step 1 is a mostly manual investigation of the library usage in the codebase that involves the code owners and domain experts. This investigation yields the relevant artifacts in the codebase, the relevant library elements, and the code patterns that describe proper library usage. In step 2, based on the results of the investigation, a custom static analysis is manually implemented using tools for static analysis and visualization. In particular, the custom static analysis captures, from the relevant artifacts in the codebase, the code fragments that match any code pattern for proper library usage. From these code fragments, the analysis extracts the relevant information and combines that information into an overview of the library usage in the codebase. Furthermore, the analysis reports on deviating library usage in the codebase: usage of the relevant libraries elements that are not matching any of the code patterns.

In the *apply* phase, the custom static analysis is executed and produces a list of deviating library usage and an overview of the library usage in the codebase. The former output contains usages of relevant library elements that do not match any code pattern for proper library usage. The latter output presents the essential information extracted from the code fragments that match any code pattern for proper library usage. The latter output is emphasized using bold in Fig. 1 to reflect the importance of the overview to enhance insight into the usage of in-house libraries.

In the *act* phase, manual action is taken based on the output of the custom static analysis. For each deviating library usage, the method requires a decision of the code owners. Based on the outcome of the decision, the method prescribes to either return to step 1 and update the specification such that the usage is considered proper library usage or execute step 3 which reduces variation in the codebase by changing the deviating library usage into a proper equivalent. Before the overview of the library usage is shown to the software engineers, also some sanity checks should be applied. Suspicious items in the overview should be verified against the codebase; if the analysis is wrong, then the method prescribes to return to step 1. Otherwise, the software engineers can use the overview and likely enhance their insight in the codebase. The software engineers' enhanced insight typically results in improvements to the codebase in step 4.

2.1. Investigate the library usage in the codebase

The goal of the investigation is to understand the library and its usage in the codebase sufficiently to create the needed custom overviews. Therefore, this investigation should determine which code patterns capture relevant information and how code fragments that match these code patterns relate to one another. Whereas the relevant information contained in the code fragments is typically domain-specific, the relations between the code fragments are often found in call graphs and inheritance trees.

The output of this investigation is a specification of the relevant artifacts in the codebase, the relevant library elements, and the code patterns capturing proper library usage. The method acknowledges that this investigation might miss some relevant details. Hence, this step will be revisited whenever the specification must be extended or updated.

We observed that discussing the following questions with stakeholders helped the investigation:

- *What are the relevant artifacts to be analyzed?* Codebases contain not only source code but also artifacts such as make files, project files, configuration files, installation scripts, test scripts, documentation, and input files for code generators. These artifacts might contain information that can focus the analysis, e.g., on the source code relevant for the specified configuration, and that can simplify the analysis, e.g., by using the input files for code generators instead of the generated code. Answering this question helps to get the relevant codebase artifacts in Fig. 1.
- *What is part of the in-house library?* We observed that it was not always clear which artifacts were part of the in-house library, especially in the following cases:
  - When similar functionality was implemented in multiple in-house libraries, such as multiple logging frameworks.
  - When wrappers around the in-house library were made, yet the library was still also accessed directly.
  - When multiple versions of an in-house library were used in the codebase at the same time.

  Yet, what constitutes the in-house library must be agreed upon by all stakeholders to ensure effective and efficient development of the custom static analysis in the next step. Answering this question helps to get the relevant library elements in Fig. 1.
- *How does the codebase interact with the in-house library?* We observed that each in-house library could support many ways of interaction, such as
  - Object-oriented interaction based on inheritance of an interface or base class provided by the in-house library;
  - Functional interface based on direct calls to functions provided by the in-house library;
  - Functional interface based on passing functions provided by the in-house library as arguments by using function pointers in C/C++ and access types in Ada;
  - Functional interface based on indirect calls to functions provided by the in-house library using messaging protocols, such as Remote Procedure Call and SOAP;
  - Test interface based on keywords used by both test code and test scripts; and
  - Implicit interaction based on exchanges of text messages with some implicit, proprietary structure.

  All kinds of interactions are relevant to determine the usage of the in-house library and its complete API. Note that the presented list is not exhaustive. For example, interactions using reflection and based on aspect-oriented techniques are possible, but were not used in our case studies. Answering this question helps to get the code patterns for proper library usage in Fig. 1.

2.2. Develop a custom static analysis

In the second step of the method, a custom static analysis for the needed overview of the library usage in the codebase is developed. The

aim of the overview is not just to enhance software engineers' insight, but also to enable them to improve the library usage in the codebase. The custom static analysis also has another output: the deviating library usage in the codebase. This output only targets the developers of the custom static analysis and is described in the next section.

The method aims for automation of the custom static analysis, since automation ensures the validity and completeness of the overview of the library usage in the codebase. Furthermore, automation enables not only the integration of the custom static analysis into the nightly-build process to keep the overview up-to-date with the codebase but also the comparison of the overviews of all branches within a source code repository. Yet, in exceptional cases, the manual execution of some custom analysis steps is more cost effective.

Software engineers can develop the custom static analysis using their programming skills in combination with tools for static analysis and visualization. The recommendations of the method for the tools, design, and overview are discussed in detail in the remainder of this section.

Tools for static analysis and visualization

The method recommends the following kinds of tools to develop the custom static analysis:

- *Parsers for the build process specifications*, such as supported by MSBuild for Microsoft Visual Studio projects; Maven for Java projects; and GNAT Components Collection for GNAT projects.
- *Parsers and semantic analyzers for the programming languages*, such as Clang and Eclipse CDT for C/C++; and ASIS and Libadalang for Ada.
- *Cross-reference databases*, such as produced by many compilers with the appropriate flag (often `xref`), and dedicated tools, such as cxref, GNATxref, and the dependency graph extractor of Renaissance for C/C++ and IDL (Dams et al., 2021) and of Renaissance-Ada (van de Laar and Mooij, 2022), and accessed either directly using query languages, like the Structured Query Language (SQL) and the Cypher query language, or indirectly via tools like GNATinspect.[10]
- *Pattern matchers for a programming language*, such as Structural Search and Replace for Java (Mossienko, 2006), the rejuvenation library of Renaissance for C/C++ (Mooij et al., 2020) and of Renaissance-Ada (van de Laar and Mooij, 2022).
- *Visualization*, of graphs, e.g., using Graphviz, Neo4j, and yEd; of tables and spreadsheets, e.g., using Excel; and of code comparisons and patches, e.g., using WinDiff.

Design of custom static analysis

The method does not consider the analysis to be a monolithic, indivisible process but rather composed of analysis steps in which the output of an analysis step can be input for other analysis steps. By splitting the analysis in multiple steps, the runtime may increase since source files may need to be processed multiple times, yet the following advantages can be obtained:

- separation of concerns;
- reuse of standard analysis steps, such as the extraction of the call graph and inheritance tree;
- enable diagnostics and validation of individual steps;
- store intermediate analysis results, e.g., to enable faster iteration and to perform regression tests; and
- reduce memory consumption.

[10] https://docs.adacore.com/gnatcoll-docs/xref.html#xref-gnatinspect.

In one of the case studies involving industrial C++ codebases, we needed to split the analysis, since, due to duplication caused by #include's, the parse trees of all C++ source files together were too large to fit in the memory of a regular laptop.

The method recommends to consider at least to split the analysis into two steps:

1. a local, intra-procedural analysis step that analyzes each compilation unit, after macro expansions, to produce nodes with local knowledge; and
2. a global, inter-procedural analysis step that captures the relations and dependencies by combining these nodes into graphs.

Similar to our suggestion, Horváth et al. (2018) generates a summary of the relevant information during the intra-procedural analysis and uses that summary to enhance the completeness of the global inter-procedural analysis.

The separation introduced by function declaration and definitions is just one of the many decoupling techniques that software engineers use to simplify the development and maintenance of codebases. Callbacks, inheritance, and broadcasting are other, well-known decoupling techniques. These decoupling techniques make codebases harder to analyze not only since more analysis steps are needed but also since without additional information, results will only be over- and under-estimations instead of exact answers.

In our method, we try to evaluate expressions at compile-time to determine the value relevant for the analysis of the in-house library. For example, the actual parameter values passed to the function calls of the in-house library can help to enhance insight in its usage. The evaluation of expressions at compile-time becomes more complicated given the earlier described two step-approach to evaluate function calls.

Overview to enhance insight

According to Reiss (2005), a visualization "is often unusable because it is overwhelming and the relevant information for a particular problem is so hard to extract". The method has the following guidelines for developing the custom static analysis to address this challenge and to produce overviews that enhance the software engineers' insight:

- Add structure to make the overview less overwhelming. For example, experiment with different structures, such as lists and trees, and with using order, e.g., sort based on alphabetic order or location in archive.
- Put the user in control to make the overview less overwhelming. For example, experiment with interactive overviews such that the user can extract the relevant information for a specific task and goal using e.g., expandable and collapsible tree structures, configurable filters, and custom queries.
- Use familiar representations to make the overview less overwhelming. For example, an overview for software engineers could use code patches, showing the differences between code fragments or between a code fragment and a code pattern.
- Experiment with multiple representations, such as text, tables, and graphs, to find the optimal overview. Textual representation includes lists of related code fragments.
- Experiment with different ways to combine data, such as grouping and aggregating based on, e.g., domain- and programming-concepts, to obtain the most relevant information.
- Relate the information in the overview to the constituent data in the code fragments for easy verification and validation of the analysis results. For example, experiment with inserting annotations and comments in source code to communicate this relation.

2.3. Check analysis and reduce variation in the codebase

As already stated, industrial codebases contain irregular code fragments. Indeed, "if programmers must obey a rule hundreds of times, then without an automatic safety net they cannot avoid mistakes" (Bessey et al., 2010). Due to such irregularities, we expect that some relevant code fragments will not match any code pattern in the initial set of proper library usage and hence the resulting overview will be initially incomplete. To ensure the completeness of the overview, the method explicitly requires to search for missed code patterns, library elements, and artifacts. To ensure the correctness of the overview, the method expects quality checks, e.g., to automatically detect conflicting information extracted by the analysis. In the remainder of this section, we describe both these checks and our recommendation to reduce the variation in the codebase.

Checks for validation

As stated in Section 2.2, the custom static analysis has besides the overview of library usage also the deviating library usage in the codebase as output. Whereas the first output is relevant for both the developers and the users, the latter output only targets the developers to ensure the validity and completeness of the overview.

Deviating library usage is found using cross-checks whenever

- a library element is used in the codebase, yet does not match any of the code patterns describing proper library usage; and
- a domain invariant is violated, for example, a stopwatch that is stopped before being started.

When deviating library usage is found, improvements to either the analysis or the codebase must be made.

Developers should also check the validity of the analysis and the overview. In the case studies, while checking unresolved references, we observed that, in the beginning, unresolved references typically indicated incompleteness of the analysis and irregularities in the codebase, yet in the end, reflected limitations of static analysis. Furthermore, we observed that in graphical overviews, checking the following nodes was typically worthwhile:

- nodes without edges, e.g., a definition without any usage;
- nodes with only incoming edges, e.g., a variable that is only written and never read; and
- nodes with only outgoing edges, e.g., a function that calls other functions, yet is never called.

In particular, we observed that, in the beginning, these nodes typically indicate incompleteness of the analysis, such as missed artifacts and relations, yet in the end, these nodes indicate possible improvements in the codebase.

Reduce variation

Industrial codebases contain variations in code, both required by the customer or the development process, e.g., to support multiple products in a single product family, and accidental since programming languages allow software engineers to realize the same functionality in multiple ways (Wall et al., 2000) . Of course, all analyses, automatic and manual, need to interpret the variations: When variations in code are due to differences in requirements that are relevant for the analysis, these variations should be treated differently. Otherwise, these variations should be treated the same.

For accidental variations, i.e., when variations in code are not due to differences in requirements, the method recommends to reduce the variation such that the codebase becomes more regular and the analysis stays simple. Of course, also all future analyses benefit from the more regular codebase.

The decision is, however, up to the code owners. When the code owners decide to keep the codebase unchanged, the complexity of the custom static analysis increases since multiple library usage variants have to be handled. Yet the method is still applicable. When the code owners decide to change the codebase, producing the custom overviews goes hand-in-hand with improving the code regularity.

2.4. Improve the codebase

The aim of the overview of library usage is not just to enhance software engineers' insight, but also to enable them to improve the library usage in the codebase. We observed that many improvements resulted from gaining overview of the library usage. To give some examples:

- The overviews showed unexpected usage of some library items. Unexpected usage includes only allocation of a library item, only read operations on a library item, and only write operations on a library item. Many of the library items with unexpected usage were considered domain-specific redundant code and were removed. In some other cases, the codebase was made more regular, e.g., by adding missing operations and test cases, to bring the usage of all library items into line with the expectations.
- The overviews showed library items and their properties together, e.g., in a single table. This clearly showed some inconsistencies between library items that went unnoticed so far due to being scattered all over the codebase. The code was improved to remove these revealed inconsistencies.
- The overviews gave insight that enabled architects to improve the interface of the in-house library and developers to refactor its code.

3. Industrial in-house blackboard library

Our first case study was executed at Philips[11] within the context of their Image-Guided Therapy systems. The case study is about an in-house blackboard library that implements a data distribution service. A blackboard conceptually consists of data items that can be accessed via getter and setter methods and using the publish–subscribe pattern. The blackboard library was introduced about 20 years ago to have a common shared state across a multitude of related modules and libraries. The blackboard library is used in 2 product lines, both the legacy product as well as the state-of-the-art product line, and in particular, by the control component that was also studied in Klusener et al. (2018). The blackboard library is not only intensively used for internal communication but also for exposing some blackboard items to external components. The whole codebase contains roughly 1 MLOC, the part that uses the library contains roughly 300 KLOC, and the library itself contains 20 KLOC.

To understand how the data flows within the control component, engineers need to understand more than just the C++ call graph. It is crucial that they also understand the data flow through the blackboard items, in particular regarding their publish–subscribe behavior. Manually obtaining this information from this large codebase is error-prone and time-consuming, making it basically impractical. This case study aims to create overviews that support software engineers in their maintenance and development work.

3.1. Investigate the library usage in the codebase

In this section we first describe an example code fragment that uses the blackboard library. Afterwards we follow the three guiding questions from Section 2.1 to obtain the relevant artifacts in the codebase, the relevant library elements, and the code patterns for proper library usage.

Fig. 2 shows a typical example for proper usage of the blackboard library:

[11] https://www.philips.com.

```
1  REGISTER(MyItem, "<some-big-uuid>")
2  EXT_PUBLISHER(MyPublisher, "<some-big-uuid>")
3  EXT_SUBSCRIBER(MySubscriber, MyItem)
4
5  class P {
6    P();
7    BooleanProxy m_proxy;
8  }
9
10 P::P(): m_proxy(MyItem) {
11   m_proxy.SetValue(true);
12   m_proxy.SetEnabled(true);
13 }
14
15 class C {
16   C();
17   HandleEvent(BoolChanged rEvent);
18   BooleanProxy m_proxy;
19   bool m_value;
20 }
21
22 C::C(): m_proxy(MyItem) {
23   m_value = m_proxy.GetValue();
24   m_proxy.Attach(*this, (BoolChanged*) nullptr);
25 }
26
27 void C::HandleEvent(BoolChanged rEvent) {
28   m_value = rEvent.GetNewValue()
29 }
```

Fig. 2. Example C++ code for a blackboard item. Horizontal lines separate the fragments that are usually located in different files.

- Line 1: use a macro to register a blackboard item with two identifiers: a human-readable variable name and a UUID;
- Line 2: use a macro to create an external publisher interface for the blackboard item (referred to by its UUID);
- Line 3: use a macro to create an external subscriber interface for the blackboard item (referred to by its human-readable variable name);
- Lines 7 and 18: define a proxy that corresponds to the type of the blackboard item's core value;
- Lines 10 and 22: initialize the proxy for the blackboard item (referred to by its human-readable variable name);
- Line 11: set the core value of the blackboard item;
- Line 12: set a meta property of the blackboard item (in this case: enabling the event handlers);
- Line 23: get the core value of the blackboard item;
- Line 24: attach an event handler for change events that corresponds to the type of the blackboard item's core value;
- Line 27–29: define an event handler to access the new value.

Relevant codebase artifacts

All blackboard items are defined within one specific C++ software component, viz., the earlier mentioned control component, which clearly needs to be in the scope of the analysis. The usage of the blackboard library is spread over 540 functions across 118 classes. In addition, there are two other parts of the codebase that need to be analyzed:

- External C++ components that access the blackboard items using the publish–subscribe pattern; and
- Test scripts that use testing keywords instead of plain C++ code to refer to blackboard items.

Initially, we overlooked the test scripts. This resulted in overviews containing many blackboard items that appeared to be useless. Once, we realized that we overlooked some relevant artifacts, we not only added these artifacts in this case study but we also improved the method to include checks to validate the completeness of the custom analysis. The checks specific for this case study are described in Section 3.3.

Relevant library elements

The main three elements of this library are:

- *Blackboard item* to store a core data value of a specific type and some meta properties that control the access by external components.
- *External publisher interface* to set the core value of a blackboard item.
- *External subscriber interface* to attach an event handler to a blackboard item.

Internally the blackboard items are accessed via proxy classes with the following functions:

- A setter function for the core data value, aka internal publisher;
- A setter function for the meta properties, e.g., to enable or disable the attached event handlers;
- A getter function for the core data value; and
- A subscription function to attach an event handler that is invoked on every change of the core data value, aka internal subscriber.

The internal interface is based on C++, and the external interface is based on Microsoft COM (Rogerson, 1997) and described using IDL.

Code patterns for proper library usage

The set of code patterns covers behavior such as:

- register a blackboard item,
- create an external publisher or subscriber interface,
- define and initialize a proxy,
- apply an operation.

Fig. 2 contains example instances of such patterns. In particular for the initialization of proxy variables, we observed a number of variations (note: not all variants are shown in Fig. 2):

- assignment to member variable:
  `m_proxy = BooleanProxy(MyItem)`
- local declaration with assignment:
  `BooleanProxy proxy = BooleanProxy(MyItem)`
- constructor chain initializer: `m_proxy(MyItem)`
- reset function from std::unique_ptr:
  `mp_proxy.reset(new BooleanProxy(MyItem))`

At several places we also observed the optional presence of additional checks:

- assert statements: `ASSERT(x != null);`
- if-statement contexts: `if (x != null){...}`

3.2. Develop a custom static analysis

In this section we first describe some of the challenges for analyzing the in-house blackboard library. Afterwards we follow the structure from Section 2.2 to describe the design of a custom static analysis.

Based on our investigation from Section 3.1 the required static analysis for the needs from Section 3 looks well-structured, but we are not aware of any off-the-shelf tool that can do it. Hence, we needed custom static analysis.

The example C++ code fragment in Fig. 2 illustrates some important aspects that need to be taken into account by any analysis of the usage of this library:

- Use of both macro (lines 1..3) and plain (lines 5..29) C++ code patterns;
- Two naming schemes for blackboard items (human-readable variable names on lines 1, 3, 10 and 22) and UUIDs (lines 1 and 2);
- Multiple proxies accessing a single blackboard item (lines 7, 10, 18, and 22);

Table 1
Tabular overview of the blackboard item `MyItem` shown in Fig. 2. The top part contains the conceptual blackboard operations, whereas the bottom part contains implementation aspects.

Operation	Primitive	Name	Location
Set core value	EXT_PUBLISHER	MyPublisher	2
Set core value	SetValue		11
Set meta property	SetEnabled		12
Get core value	GetValue		23
Attach event handler	EXT_SUBSCRIBER	MySubscriber	3
Attach event handler	Attach		24
Initialize proxy		P::m_proxy	10
Initialize proxy		C::m_proxy	22
Define event handler		C::HandleEvent(BoolChanged)	27

- Selecting the event handler based on the change event type (line 24); and
- Code decoupling in multiple functions and files. The code decoupling over files is visualized in Fig. 2 using horizontal lines.

Tools for static analysis and visualization

We used the following tools:

- *Parser for the C++ programming language:* Eclipse CDT for C/C++.
- *Cross-reference database:* The dependency graph extractor of Renaissance for C/C++ and IDL.
- *Pattern matcher for the C++ programming language:* The rejuvenation library of Renaissance for C/C++ (Mooij et al., 2020).
- *Graph visualization:* Neo4j and yEd.

Design of custom static analysis

We have incrementally extracted the required data using the following steps:

1. Process all relevant code fragments in the main C++ component, and relate them to global blackboard items;
2. Use a cross-reference database to find usages of external subscribers and publishers in other C++ components; and
3. Use a lexical analysis to find usages in the test scripts.

We have analyzed all C++ files in isolation without expanding included files nor using specific build configurations. We did not need any custom analysis of the IDL files, as all IDL identifiers (UUIDs) can be obtained from the analyzed C++ code.

We have used limited forms of data flow analysis to deal with brackets and conditional expressions, such as in the following statement:

```
(... ? m_proxy_left : m_proxy_right).SetValue(...);
```

For a few exceptional cases we have decided to perform the analysis manually instead of extending the automated custom static analysis, because of the expected required efforts.

Overview to enhance insight

We have created multiple views on the extracted data, differing in aspects like:

- representations (textual, tabular, and graphical);
- subsets of the data;
- grouping (by blackboard item and by C++ class); and
- aggregation (at class-level and at function-level).

We give two examples of overviews that were used within this case study. The content shown in these examples corresponds to the code fragment from Fig. 2.

- Table 1 shows a tabular overview with all extracted data grouped by blackboard item. For each blackboard item, it gives a specific list of details. Such a tabular representation is easy to generate, and suitable for collecting heterogeneous data about individual blackboard items. In one glance it gives an overview of how a blackboard item is used across multiple files.
- Fig. 3 shows a graphical overview with only the main library operations aggregated at class-level. In one glance it gives an overview of the relations between blackboard items; their external publishers and subscribers; and classes that internally access them. Such a graphical overview helps to see relations between blackboard items, and to reason about how data can flow through the system.

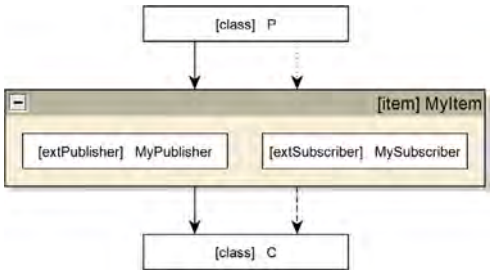

Fig. 3. Graphical overview in yEd of the blackboard item shown in Fig. 2. This overview shows the blackboard item `MyItem`, the external publisher `MyPublisher`, the external subscriber `MySubscriber`, and the classes `P` and `C` that internally publish and subscribe to the blackboard item via a setter and getter function, respectively. Edges represent publish and subscribe (solid), meta-setter (dotted), and getter (dashed) functions.

When developing code using this blackboard library, it is difficult to keep an overview of where and how each blackboard item is used. Table 1 shows an overview of blackboard item `MyItem` from Fig. 2. In addition to the main library operations (e.g., set core value) we also include some implementation aspects (e.g., initialize proxy, and define event handler) that help to find all code fragments related to a blackboard item. The column "primitive" mentions the specific library element or function that is used, and the column "name" the new name that is introduced (if any). Finally the table includes the location (line number) in the code.

Table 1 shows a global aspect of the source code that is difficult to grasp in traditional ways. The small code fragment from Fig. 2 would normally be distributed over 5 files: the two classes P and C have their own header and implementation file, and in addition there is a file that contains the macro's that define the blackboard item and external publishers and consumers. In practice there are more blackboard items, more files, and typically more table entries per blackboard item. The overview table clearly indicates how and where specific blackboard items are used.

We have also combined the blackboard information with call graphs, which is useful as both method calls and publish/subscribe patterns represent data flow. Thus we obtain a more complete overview of how the data flows through the control component of the Image-Guided Therapy systems.

3.3. Check analysis and reduce variation in the codebase

Along the lines of Section 2.3, we describe the ways we have checked and improved the completeness of the custom analysis.

Checks for validation

We have performed checks based on the created overviews by looking for blackboard items with suspicious usage:

- Blackboard items that are not used;
- Blackboard items that are only written or only read;
- Blackboard items with meta property writes for external subscribers, but that have no external subscribers;

Table 2
Removal of domain-specific redundant code.

	Initial		Removed		Final	Reduction
Blackboard items	273	–	108	=	165	39 %
External publisher interfaces	50	–	19	=	31	38 %
External subscriber interfaces	95	–	31	=	64	32 %

- Blackboard items with meta property writes for external publishers, but that have no external publishers.

For the blackboard items with suspicious usage, we have searched for potentially relevant code fragments that the analysis may have missed. The instances for which we could not find such code fragments have been discussed with the software engineers of the codebase. In particular, this led to the discovery mentioned in Section 3.1 that we needed to include test scripts in our custom static analysis.

Reduce variation

In this first case study we had not yet identified the step of reducing variation in the codebase. Moreover, the variations discussed in Section 3.1, such as optional assert statements, could easily be handled by the Renaissance pattern matching.

3.4. Improve the codebase

The last step, from Section 2.4, focuses on how the overviews can be used to improve the codebase.

The industrial software engineers were interested in the overviews of how and where the blackboard items are used. They recognized that the overviews provide a wealth of information that can often be used when reasoning about the interaction between components.

In particular the blackboard items with suspicious usage from the checks indicated opportunities for improving the codebase. It turned out that no-longer-used blackboard items were not always completely removed from the codebase: these blackboard items with suspicious usage were, in fact, the remainders. Modern compilers warn about programming language items, e.g., variable and function declarations, that are declared but never used. Domain experts however want warnings for blackboards items that are not both written and read. Our analysis provides these warnings by using more domain knowledge about the blackboard items.

Based on our automated analysis, we manually removed the domain-specific redundant code. Table 2 summarizes the realized removal. The removed external interfaces were no longer used by any external component.

During our case study, it turned out that the engineers wish to reduce the use of blackboard items. Our analysis provides an overview of the current usage, in which we observe that some blackboard items (for example, those without subscribers) could be replaced by a standard variable. Our analysis also shows that a specific group of C++ classes is involved in many blackboard items and deserves further human analysis. The implementation of these observations is outside the scope of this work.

3.5. Lessons learned

The first case study was the starting point of our method from Section 2. Primarily it has shown the feasibility of extracting valuable overviews of the usage of in-house libraries using custom static analysis. It has also illustrated that the method needs checks to validate the custom analysis and its output, the overview of library usage.

As described in Section 3.2 we have not automated a few exceptional cases. This illustrates a crucial point in our line of reasoning, viz., to take into account the effort required to automate analysis steps. In later cases we have expanded this thought in the direction of reducing the variation in the codebase.

4. Industrial in-house logging library

Our second case study focuses on the analysis of an in-house logging library. Via this library, the embedded software of the Image-Guided Therapy systems of Philips[12] logs messages describing the components' behavior and interaction. These log messages can be used for documentation, debugging, and diagnostics. This case study focuses on the state-of-the-art product line with 6 codebases, that together contain roughly 6 MLOC.

Each log message consists of two parts:

- an event type, with attributes like ID, description, and severity; and
- additional attributes.

The following design guidelines are assumed on the log message specification:

- An event type is unique per component and contains immutable information.
- An event type is associated with a fixed set of additional attributes.
- The textual representation of the additional attributes is based on key–value pairs. The 'key' is immutable and unique for that attribute. The 'value' is variable. A key–value pair should be logged as "['key' = 'value']".

However the logging library offers no way to enforce these design guidelines. Without a safety net, mistakes are expected (Bessey et al., 2010). Indeed, developers observed that, over the years, unwanted variations in the code (patterns) generating logging and in the logged messages have appeared, complicating managing and interpreting logging. For instance, the maintenance of the tools that post-process the generated logs has become increasingly difficult and developers lost a general overview of all the logging that could be generated by the code.

To get insight into the log messages that can be generated by Philips code, we create an overview collecting all the information known at compile time, such as event types and keys of additional attributes. The information known at runtime, e.g., the values associated with the additional attributes, is not in the overview. This overview is sufficient to understand all the log messages and verify if they adhere to the aforementioned prescribed design guidelines. Additionally it can be used in reasoning about redesigning the logging library.

4.1. Investigate the library usage in the codebase

Multiple components produce log messages within the analyzed Philips system. Each component contains between hundred thousands and millions of lines of C++ and C# code. In the remainder of this section, we only report examples from the C++ codebase. Additionally, multiple logging libraries were used in the different components using different programming constructs to implement the concepts of log messages. We will report only one of these cases exemplified in Fig. 4.

Fig. 4 shows how two log messages, related to the event types `eA` and `eB` defined on lines 2 and 3, can be produced by the code. The function `Func` (on lines 19 to 22), calls the `LogSetOn` function on class `C` with an event type depending on its argument `b`.

Within `LogSetOn`, on lines 14 to 17, a call to the logging library API `evLog` generates the actual logging and appends additional attributes (line 16). The wrapper function `CreateInfo` creates two instances of key–value pairs for the additional attributes in the correct format, by calling the logging API `additionalInfo`, and creates a `"free-format string"`, violating the design guidelines.

In this example, we see variations in how the code is written. For example, the key of an additional attribute can be a string literal,

[12] https://www.philips.com.

```
1  const int iBase = 100;
2  const Event eA = { iBase + 1, "EventA", WARNING };
3  const Event eB = { iBase + 2, "EventB", ERROR };
4
5  const string ClassTag = "Class";
6
7  string C::CreateInfo(string s) {
8    string a = Log::additionalInfo("State", s);
9    a += Log::additionalInfo(ClassTag, "C");
10   a += "free-format␣string";
11   return a;
12 }
13
14 void C::LogSetOn(const Event& event) {
15   Log log;
16   log.evLog(event) << CreateInfo("On");
17 }
18
19 void Func(bool b) {
20   C c;
21   c.LogSetOn(b ? eA : eB);
22 }
```

Fig. 4. Example C++ code fragment for logging.

e.g., `"State"` on line 8, and a variable, e.g., `ClassTag` on line 9. This variation makes the code's maintenance less practical and the analysis more complicated, i.e., requiring data flow analysis in the latter case to determine its value. We also see variation in the format of the logged additional attributes. For example, the prescribed format of the logged additional attributes is used on lines 8 and 9, but not on line 10. This variation makes the post-processing of the logged messages unnecessarily complex.

Relevant codebase artifacts

In the first step of our method, as described in Fig. 1, we have analyzed the codebase, understanding how the logging happened per unit. From the Visual Studio Solution file, we extracted the relevant Visual Studio Project files. From the Visual Studio Project files, we extracted the relevant code files. We only analyzed these code files, leaving out documentation, tests, and test input.

Relevant library elements

We will describe the relevant library elements by referring to the example of Fig. 4. A log event type is created via a 'struct' declaration (cf. lines 2 and 3) and can be logged via a dedicated method (`evLog`) of the `Log` class (cf. line 16). Additional attributes are created, preferably (but not exclusively) via a dedicated API (cf. `Log::additionalInfo` on lines 8 and 9), and appended to the logged message via an overloaded append operator (cf. operator << on line 16).

Code patterns for proper library usage

The set of code patterns to detect the usage of the library includes:

- a pattern for the event type declaration;
- a pattern to detect calls to the method `evLog` of the `Log` class;
- a pattern to detect calls to the method `additionalInfo` of the `Log` class; and
- a pattern to detect the overloaded append operator of the additional attributes.

4.2. Develop a custom static analysis

To help Philips to get insight in all the types of log messages that the system can generate, we generated an overview, as shown in Table 3. This overview contains all the possible logging events ordered by identifiers and their additional attributes. This overview concentrates, in a single place, information that is otherwise scattered throughout the code and provides a means to identify possible data inconsistencies, unwanted variations, and unwanted changes to log messages.

The example of Fig. 4 shows some challenges with which the analysis needs to deal:

1. Event IDs may contain simple arithmetic; thus, they require a compilation-time arithmetic expression evaluation (lines 2 and 3).
2. Appended information does not always adhere to the prescribed format and requires a compilation-time string evaluation (line 10).
3. Logging code may be decoupled in multiple functions, which declaration, definition, and usages span multiple files as is visualized in Fig. 4 using horizontal lines.
4. Natural growth of the code brings a proliferation of wrappers and code (patterns) for logging. This is not shown in the picture for sake of simplicity.

The used tools, design, and overviews to address these challenges are described in the remainder of this section.

Tools for static analysis and visualization

The analyses for C++ and C# use different tools, namely Eclipse CDT and Roslyn, respectively. Therefore, the analyses have been developed separately, yet are based on the same design principles.

We used the following tools:

- *Parser of Visual Studio Solution and Project files:* MSBuild.
- *Parser for the C++ programming language:* Eclipse CDT for C/C++.
- *Parser for the C# programming language:* Roslyn.
- *Pattern matcher for the C++ programming language:* The rejuvenation library of Renaissance for C/C++ (Mooij et al., 2020).

Design of custom static analysis

To handle challenges 1 and 2 of the beginning of this section, we use compile-time evaluation of integers for the event IDs and strings for the additional attributes. Event IDs and additional attributes may use variables whose definitions are scattered through the code and require data flow analysis to be resolved. Moreover, the key–value pairs of the additional attributes have no dedicated type (and could be any string without an enforced format). In many cases, they were composed by combining many string variables. Therefore their evaluation initially resulted in the Cartesian product of all possible values of those string variables leading to an exponential explosion of the number of final possible values, i.e., entries in the final overview. This fact rendered the final overview unusable.

To solve this problem, we limited the evaluation of particular string variables to stop the data-flow analysis. The expression evaluation algorithm returns with a wildcard character, indicating that the value of the string at that point is not crucial for the overview. For instance, we would stop the evaluation of variables containing file names in logs reporting an error occurring in a file. So instead of having hundreds of occurrences of the same error message on different files, we would have only the generic, parameterized error message "Error in file %s" indicating what type of logging message was generated from that location in the code. Another example of such a filter is shown in Fig. 4 on line 10 and in the corresponding entries in Table 3. Here all the strings generated by the function `CreateInfo`, which are not produced using the `Log::additionalInfo` API, are presented in the overview using the key %s to signal free-formatted text.

To handle challenge 3 of the beginning of this section, we have split our custom static analysis into a two-step approach:

1. a time-consuming step that processes the declarations in each compiled file (with the expansion of includes); and
2. a quick step that integrates the results from the previous step and, in particular, deals with decoupling patterns, starting from the `evLog` calls.

The final overview and the intermediate data are labeled with source locations to facilitate debugging and analysis.

To handle challenge 4 of the beginning of this section, we had to choose between only analyzing the standard library usage, somehow

Table 3
Tabular overview for the example from Fig. 4.

ID	Description	Severity	Appended information
101	EventA	Warning	State, Class, %s
102	EventB	Error	State, Class, %s

working our way through the wrapper code, or analyzing selected wrappers as alternative logging mechanisms. However, the first choice was not always possible, especially for logging messages across server–client process boundaries. Running the analysis for the logging information takes up to a couple of hours in the most significant unit with a million code lines. The analysis runtime is of the same order of magnitude as code compilation. This runtime is considerable but was deemed acceptable to include this analysis in the nightly sanity-check build.

Overview to enhance insight

Table 3 gives an example overview corresponding to the C++ code fragment of Fig. 4. The generated overviews were similar to Table 3, yet contained over a hundred rows. These overviews fostered good discussions between the various engineering teams, ranging from the requirements on logging to the intended way of using the in-house library. For this purpose, the overview abstracts irrelevant details and captures cross-cutting aspects of the source code that are difficult to grasp in other ways. For instance, in a list representation of all the log messages, it was relatively simple to see if and how the same event ID was used with different sets of additional attributes. In contrast, establishing this manually would require a complex, error-prone, analysis to resolve the keys of the additional attributes. In addition, the overviews indicate inconsistencies between related event types. For instance, in a specific code unit with an old and a new version of the code maintained simultaneously, the log event identifiers of the old and new versions were not consistent, revealing a so far undetected fault.

4.3. Check analysis and reduce variation in the codebase

The following variations were removed to make the codebase more regular and to keep the analysis simple.

- An unnecessarily complex way to initialize a map of events indirectly via a constructor was replaced by a simple initialization.
- Violations of the design guideline that information of event types is immutable were removed by rewriting the code to use multiple event types instead of reusing and mutating a single event type.
- A switch statement with an unreachable default case that logged an ill-formatted event was rewritten to log a well-formed event.

Checks for validation

We use two primary checks at two different levels:

- a local cross-check that reports `evLog` calls for which the analysis fails to resolve the event or additional attributes, and
- a global cross-check where the analysis fails to find usage of a relevant parameter or function in the codebase.

During the construction of the custom static analysis we also looked at other checking mechanisms to make the produced overview as complete as possible. Some of those checks have been promoted to feedback for users once we were sufficiently confident of the quality of the produced overview. For example, a check that looks for event declarations that are not used in any `evLog` statement. Initially, this check indicated the incompleteness of our custom static analysis. But after iteratively improving our analysis, this check reported unused events, i.e., domain-specific redundant code, in the analyzed codebase.

Two other additional checks were realized in post-processing to validate the analysis result.

- A cross-check against a logging design specification, where that existed.
- A cross-check against the logging results of tests.
  The comparison against test results was automated with Python scripts. The comparison showed that:
  - As expected, the tests trigger fewer logging events than those detected by static analysis.
  - All log messages generated during testing were also detected by our static analysis, either as entries in the generated overview or reported by the cross-checks.
  - Entries not adhering to the prescribed format of the logging design, had counterparts in the set of logs generated by tests, for instance, log event descriptions with wildcards %s appeared as occurrences of logging with the same event log ID but different descriptions.

4.4. Improve the codebase

Based on the overviews, the following code improvements were identified.

- *Remove unused events:* We detected many event declarations that were only used in a function that reports all event declarations. Some of them were reported in separate open issues in the issue-tracking system as unused log events. Based on the overview, all these event declarations were removed and marked as obsolete to avoid accidental reuse.
- *Solve inconsistencies:* Some small errors (capitalization, copy-paste-not-modified, inconsistent IDs) in log event types and `additionalInfo` format have been solved.
- *Standardize usage:* At many places, the set of additional attributes was made unique per event to adhere to the design requirements.

Note that these specific issues would not have been found with off-the-shelf static analyzers.

In addition, the result of the provided analysis was used for the subsequent re-design of the logging framework. We used an approach based on a domain-specific language (DSL) to generate the implementation of new logging APIs. A single, well-designed logging framework was adopted for all the units, and the format of the additional attributes was enforced in the new APIs. The results of the analysis tool were used to extract information to generate the APIs' DSL specification and refactor the code using the newly designed APIs.

4.5. Lessons learned

This case showed us the importance of a re-usable method to quickly build custom analyses for the different logging frameworks of the various code repositories. For the first time, we have validated the results of our method against system test results. This confirmed that, as expected, static analysis covers more logging than tests. Thanks to this case, we enriched our C++ and C# rejuvenation library with data-flow analysis and compile-time evaluation of simple arithmetic expressions and strings. The compile-time string evaluation brought an exponential explosion in the number of entries in the logging overview. To limit the explosion, we terminated the string evaluation based on case-specific conditions related to problematic cases, e.g., cases producing many repetitions of almost the same logging message with slight variations such as file names.

5. Industrial in-house inspection library

Our third exploratory case study was executed at ITEC,[13] an independent subsidiary of Nexperia.[14] ITEC is an equipment and automation

[13] https://www.itecequipment.com.
[14] https://www.nexperia.com.

partner of semiconductor manufacturers. The equipment produced by ITEC guarantees the quality of both the production process and the final product by performing vision-based inspections. A vision-based inspection controls cameras and (flash) lights to capture images and analyses them. ITEC currently realizes the vision-based inspections using two in-house inspection libraries.

Since 2005, the basic inspection library is under development by ITEC. This library contains 20K lines of Ada code and is an essential part of the complete codebase of 1.5M lines of Ada code. However, in 2017, ITEC concluded that the basic inspection library was at the end of its life cycle. Since then, the flexible inspection library is under development.

Despite that the flexible inspection library was still under construction, the development of just a few vision-based inspections using the flexible inspection library convinced ITEC to migrate all vision-based inspections from the basic to the flexible inspection library. They did not intend to automate this migration fully but preferred manual changes whenever they estimated that automation would not reduce the overall needed time and effort.

Our third case study was initiated to support this migration. Yet, the goal of the third case study was not to perform the migration but to enhance insight into the usage of the basic inspection library within the ITEC codebase: not only a prerequisite for changing the codebase but also for the associated effort estimation and planning. To be able to describe what ITEC needed, we first have to explain the abstractions that the basic inspection library provides in the domain of vision-based inspections.

The basic inspection library supports the following three domain-specific concepts:

- *Inspection process (85 instances):* analyzes a (part of a) camera image.
- *Inspection container (25 instances):* orchestrates all inspection processes that analyze the same camera image.
- *Inspection location (8 instances):* orchestrates all inspection containers that share the same camera and associated (flash) lights to obtain camera images.

A vision-based inspection contains at least one inspection process. A vision-based inspection is not limited to a single inspection container or location. For example, a vision-based inspection might check a product by using camera images that are made at different locations with different (flash) light conditions.

To support the migration, ITEC needed to know

- How does each inspection process, container, and location use the library?
- Where are inspection processes, containers, and locations created?
- How is each inspection process, container, and location configured? I.e., which configuration values are used?
- Where and how are the settings of each inspection process, container, and location changed?
- Where and how are functions of the library called?
- Which inspection process, container, and location are involved in every function call to the library?

5.1. Investigate the library usage in the codebase

To enhance insight necessary to support the migration from the basic to the flexible inspection library, we started with the first step of the proposed method, as depicted in Fig. 1, and manually investigated, together with ITEC experts, the usage of the basic inspection library in the ITEC codebase.

The investigation showed that the usage of the basic inspection library is distributed across the codebase: Not only many vision-based inspections, but also many inspection processes, containers, and locations regularly are scattered over multiple functions and files. Furthermore, the investigation showed that the usage of inspection processes, containers, and locations is similar throughout the codebase, so we will focus in the example and discussion on the usage of inspection processes in the remainder of this section.

```
1  procedure Init_Inspection_Location
2    (il : Inspection_Location)
3  is
4  begin
5    Set_Process (il.Container (8),
6      Template_Type, 1, "Transfer_1_Align");
7    Set_Process (il.Container (8),
8      Circle_Type, 2, "Transfer_1_PinA1");
9    Set_Process (il.Container (9),
10     Line_Type, 1, "Transfer_2_Edges");
11 end Init_Inspection_Location;
12
13 procedure Run_Inspection_Location
14   (il : Inspection_Location)
15 is
16   psps : Procs_Params;
17 begin
18   Get_Procs_Params (il.Container (8), psps);
19   psps.Par (Match_Kind)(1).ROI_Grid := Shifted;
20   psps.Par (Ruler_Kind)(2).ROI_Grid := Absolute;
21   Set_Procs_Params (il.Container (8), psps);
22
23   Get_Procs_Params (il.Container (9), psps);
24   psps.Par (Ruler_Kind)(1).ROI_Grid := Relative;
25   Set_Procs_Params (il.Container (9), psps);
26 end Run_Inspection_Location;
27
28 procedure Handle_Inspection_Location
29   (il : Inspection_Location)
30 is
31 begin
32   Init_Inspection_Location (il);
33   Run_Inspection_Location (il);
34 end Handle_Inspection_Location;
```

Fig. 5. Example.adb: Ada code using the basic inspection library.

Fig. 5 shows the usage of the basic inspection library in a simplified, yet representative Ada file. Like in the ITEC codebase, magic numbers, in particular, `1`, `2`, `8`, and `9`, are used to identify instances of inspection processes and containers. The procedure `Init_Inspection_Location` is defined on lines 1–11. On lines 5 and 6, the procedure `Set_Process` sets the type and name of inspection process `1` within the inspection container `il.Container (8)` to `Template_Type` and `"Transfer_1_Align"`. Similarly, lines 7 and 8 set the type and name of inspection process `2` within the same inspection container (`il.Container (8)`). Finally, lines 9 and 10 set the type and name of inspection process `1` within another inspection container (`il.Container (9)`). Note that lines 5 and 6 and lines 9 and 10 refer to different inspection containers, so although the same identifier (`1`) is used, these identifiers refer to different inspection processes.

The procedure `Run_Inspection_Location` is defined on lines 13–26. On line 18, the procedure `Get_Procs_Params` assigns the parameters of all processes of the inspection container `il.Container (8)` to the local variable `psps`. Line 19 assigns `Shifted` to the process parameter `ROI_Grid` of inspection process `1` within the local variable `psps`. Note that also the kind of the inspection process is used to access the process parameter. Similarly, on line 20, `Absolute` is assigned to the process parameter `ROI_Grid` of inspection process `2`. On line 21, the procedure `Set_Procs_Params` sets the parameters of all processes of the inspection container `il.Container (8)` to those of the modified local variable `psps`. Note that the process parameters of both inspection processes are changed together. Lines 23–25 similarly assign `Relative` to the process parameter `ROI_Grid` of the inspection process `1` within the inspection container `il.Container (9)`.

Finally, the procedure `Handle_Inspection_Location` is defined on lines 28–34 and calls the previously described procedures to ensure an inspection location is initialized before being used.

Table 4
Created inspection containers.

ID	Variable
CA	il.Container (8)
CB	il.Container (9)

Table 5
Created inspection processes with their configuration values.

ID	Name	Kind	Type	Location
CA.1	Transfer_1_Align	Match	Template	5:3–6:42
CA.2	Transfer_1_PinA1	Ruler	Circle	7:3–8:40
CB.1	Transfer_2_Edges	Ruler	Line	9:3–10:38

Relevant codebase artifacts

For the analysis, only the Ada source code files were considered relevant. Hence, the analysis included all Ada source code files. All Ada source code files were quickly scanned, and those files in which the basic inspection library was used or implemented were analyzed in more detail.

Relevant library elements

The relevant library elements in the codebase were in correspondence with the domain-specific concepts of inspection process, inspection container, and inspection location. The information related to a library element was unfortunately not localized but scattered throughout the codebase. In the example, line 19 contains additional information (its kind) of the inspection process set on lines 5 and 6.

Code patterns for proper library usage

Based on our investigation of the codebase together with the ITEC experts, the code patterns for proper library usage included

- a pattern that matches setting an inspection process, resulting in the example of Fig. 5 in three matches within the procedure `Init_Inspection_Location`;
- a pattern that matches the changing of the parameters of an inspection process, resulting in the example of Fig. 5 in three matches within the procedure `Run_Inspection_Location`;
- a pattern that matches the declaration and initialization of an inspection process as separate statements;
- a pattern to select one of the 85 inspection processes; and
- a pattern to change a parameter of the currently selected inspection process.

As already stated, at the start of our case studies it was not clear how to determine the regularity of a codebase. So, we had to observe experimentally how regular the ITEC codebase is with respect to these custom code patterns.

5.2. Develop a custom static analysis

The goal of the custom static analysis in this case study is to enhance insight into the usage of the basic inspection library within the ITEC codebase that is vital to estimate, plan, and perform the migration to the flexible inspection library. For the migration of the inspection processes as shown in the code of Fig. 5, the information as contained in the tabular overview shown in Tables 4, 5, 6, and 7 is needed. We will briefly describe each table before we describe the challenges to extract this information from the codebase.

Table 4 contains all extracted inspection containers. Since the example only focuses on inspection processes, the table only contains for each inspection container an identifier and the related variable. Note that these identifiers are not extracted from the code but are introduced to simplify the other tables.

Table 6
Calls to the basic inspection library.

Context ID	Callee	Caller	Location
CA	Get_Procs_Params	Run_Inspection_Location	18:3–17:44
CA	Set_Procs_Params	Run_Inspection_Location	21:3–21:44
CA.1	Set_Process	Init_Inspection_Location	5:3–6:42
CA.2	Set_Process	Init_Inspection_Location	7:3–8:40
CB	Get_Procs_Params	Run_Inspection_Location	23:3–23:44
CB	Set_Procs_Params	Run_Inspection_Location	25:3–26:44
CB.1	Set_Process	Init_Inspection_Location	9:3–10:38

Table 7
Changes of parameters of inspection processes.

Context ID	Parameter	Value	Location
CA.1	ROI_Grid	Shifted	19:3–18:47
CA.2	ROI_Grid	Absolute	20:3–19:48
CB.1	ROI_Grid	Relative	24:3–24:48

Table 5 contains all inspection processes. For each inspection process, the table contains its identifier, name, kind, type, and location where it is set. The globally unique identifier is the combination of the inspection container identifier with the locally unique process identifier as used in the code. Note that this table combines information from multiple locations. For example, the name and kind of inspection process 'CA.1' can be observed on line 6 and 19, respectively.

Table 6 contains all calls involving an inspection container or process. For each call, the table contains the identifier of the involved inspection container or process, the called procedure, the calling procedure, and the call location. Note that since the procedures `Init_Inspection_Location` and `Run_Inspection_Location` are called with the same argument from the procedure `Handle_Inspection_Location`, the calls within different procedures still refer to the same inspection containers.

Finally, Table 7 contains all changes to any parameter of the inspection processes. For each set operation, the table contains the identifier of the inspection process, parameter, assigned value, and location.

The challenges to extract the information from the codebase include

- combining information scattered over files and procedures throughout the codebase;
- finding particular patterns in the codebase;
- relating a parameter change to a specific inspection process, although the changes of parameters of multiple inspection processes are grouped together;
- relating the values passed in function calls to the corresponding parameters within the called functions; and
- handling the flow of information through local variables.

We executed the second step of the proposed method, as depicted in Fig. 1, to develop a custom static analysis and address these challenges as described in the remainder of this section.

Tools for static analysis and visualization

We used the following tools:

- *Parser for the build process specifications of GNAT projects:* GNAT Components Collection.
- *Parser and semantic analyzer for the Ada programming language:* Libadalang.
- *Pattern matcher for the Ada programming language:* The rejuvenation library of Renaissance-Ada (van de Laar and Mooij, 2022).

Design of custom static analysis

Initially we developed a monolithic solution. Yet in a later iteration, we separated the analysis to better cope with its complexity. In particular, we split our custom static analysis in two consecutive processes:

1. *Local analysis:* The analysis collects for each procedure information about not only calls and their arguments, but also all matches of proper code patterns. When possible, this information is expressed using the parameters of that procedure.
2. *Global analysis:* The domain-specific data for migration is extracted, among others by following the hierarchy of procedure calls and by replacing the parameters of a procedure by their actual values to simplify the information.

Overview to enhance insight

We did experiments with different ways to share the overview with the ITEC experts. We tried a direct representation of the overview of the code that uses tables, such as Tables 4, 5, 6, and 7. We also tried some indirect, code-oriented representations:

- Add comments, around the code fragments that use the library, that contain information, such as the involved inspection process, container, and location, or provided links to related code fragments; and
- use the codebase after migrating a specific inspection process, container, or location, towards the flexible inspection library.

For both cases, code comparison with the original codebase shows all related code fragments and thus also provides an overview of library usage in the codebase. Fig. 6, that shows the automatically migrated code of the example code in Fig. 5, is an example of the latter. The ITEC experts stated they already knew the indirect representations from other software-development activities, such as reviewing and analyzing code. In particular, the ITEC experts preferred the patch of the codebase . We assume that besides being a known representation also performing the migration was considered valuable.

5.3. Check analysis and reduce variation in the codebase

Along the lines of Section 2.3, we describe the ways we have checked and improved the custom analysis.

Checks for validation

Although we did some manual checking before presenting our overviews to ITEC, most checks were automated. In particular, we developed the custom static analysis to provide a list of deviating library usages in the ITEC codebase, using the following checks:

- check whether each declared inspection process is initialized,
- check whether all references to "Set_Process" are found and analyzed,
- check whether all references to "Set_Procs_Params" are found and analyzed,
- check whether each inspection location has at least one inspection container, and
- check whether each inspection container has at least one inspection process.

Executing the custom static analysis on the ITEC codebase produced only a short list of deviating library usages. Hence, the ITEC codebase was quite regular. Most deviating library usages were related to the code pattern that matches the declaration and initialization of an inspection process as separate statements. This code pattern had 13 'violations' where the declaration and initialization of an inspection process were combined in a single statement.

Reduce variation

We addressed the deviating library usages in a few cases by extending the proper code patterns to capture this variation. Yet in most cases, we addressed them by reducing the variation in the ITEC codebase. For example, the 13 'violations' were removed by splitting the combined declaration and initialization into two statements. An additional benefit of making the codebase more regular is that the custom static analysis was kept as simple as possible.

```
procedure Init_Inspection_Location
  (il : Inspection_Location)
is
begin
  declare
    Insp : constant Match_Specifics_Access :=
      Match_Specifics_Factory.
        Create ("Transfer_1_Align");
  begin
    il.Container_IF (8).Add_Inspection (Insp);
    Transfer_Inspections (Transfer_1_Align)
      := Insp;
  end;
  Set_Process (il.Container (8),
    Circle_Type, 2, "Transfer_1_PinA1");
  Set_Process (il.Container (9),
    Line_Type, 1, "Transfer_2_Edges");
end Init_Inspection_Location;

procedure Run_Inspection_Location
  (il : Inspection_Location)
is
  psps : Procs_Params;
begin
  Get_Procs_Params (il.Container (8), psps);
  Set_ROI_Grid
    (Match_Specifics_Access
      (Transfer_Inspections
        (Transfer_1_Align)).Teach.all,
     ROI_Grid => Shifted);
  psps.Par (Ruler_Kind)(2).ROI_Grid := Absolute;
  Set_Procs_Params (il.Container (8), psps);

  Get_Procs_Params (il.Container (9), psps);
  psps.Par (Ruler_Kind)(1).ROI_Grid := Relative;
  Set_Procs_Params (il.Container (9), psps);
end Run_Inspection_Location;

procedure Handle_Inspection_Location
  (il : Inspection_Location)
is
begin
  Init_Inspection_Location (il);
  Run_Inspection_Location (il);
end Handle_Inspection_Location;
```

Fig. 6. Ada code after migration of the inspection process 'CA.1' to the flexible inspection library.

5.4. Improve the codebase and its development

Executing the custom static analysis on the ITEC codebase produced an overview of the library usage, similar to Tables 4, 5, 6, and 7. In step 4, we used that overview to improve the library usage in the codebase by migrating usage from the basic to the flexible inspection library. Although we could migrate all inspection functionality at once, ITEC wanted to migrate the inspection functionality in a number of iterations for the following reasons:

- to reduce the risk of migration,
- to minimize the manual test activities involved,
- to ensure easy diagnosis and root-cause analysis in case of unexpected results, and
- to enable learning by using the newly obtained insights to improve the analysis, the extracted overview, and the automatic part of the migration.

So in each iteration, we focused on one part of the overview at a time. In the first iterations, we focused on a single inspection process. Fig. 6 shows the code after migrating a single inspection process of our running example as shown in Fig. 5. In later iterations, we were confident that inspection processes were correctly migrated and we expanded the focus, e.g., to a single inspection container.

In all iterations, the activities we performed were:

- Manually select the part from the overview relevant in this iteration.

- Automatically transform the ITEC codebase using the selected part of the overview.
- Discuss the selected part of the overview together with the transformed code with the ITEC experts.
- When needed update the extracted overview and the ITEC codebase based on the feedback of the ITEC experts.
- Make some minor manual changes to compile the ITEC codebase.
- Test the migrated part.
- When issues were found, they were not only solved but also used as feedback to improve the analysis.
- Commit the final code to the ITEC codebase.

In all iterations, new insights with respect to the ITEC codebase were obtained. Most insights were obtained while discussing the selected part of the overview together with the transformed code with the ITEC experts. Most obtained insights were related to the changes needed to migrate a part: the overview exposed functions that were initially not considered by the ITEC experts to be included in the migration.

After all iterations, the basic inspection library was no longer used and could be removed from the ITEC codebase. This removal severely simplified the maintenance activities of the inspection functionality. Finally, we want to mention that

- although the migration was already challenging on itself, keeping it synchronized with the large number of ongoing developments added an additional layer of complexity; and
- the enhanced insight into the inspection functionality even resulted in improvements of the flexible inspection library.

5.5. Lessons learned

1. Initially we developed a monolithic solution. Yet in a later iteration, we separated the analysis in two parts to better cope with its complexity.
2. For our analysis we had to deal with (extreme) decoupling: the analysis needed global context from multiple earlier nested method calls.
3. An iterative way-of-working was necessary, since initially the goal of the analysis was not completely clear. When the first insights were obtained, the goal of the analysis became clearer, yet also shifted slightly.
4. We had multiple ways of communicating our analysis results with the ITEC experts. The ITEC experts preferred the patch of the codebase . We assume that besides being a known representation also performing the migration was considered valuable.
5. We decided not to migrate all inspection functionality using the basic inspection library at once but to use a number of iterations. Since the details of the overview, such as line numbers, changed in every iteration, we had to run our analysis at the start of each iteration. This added a new requirement to our custom static analysis since some code patterns also matched the migrated code: exclude already migrated code from the overview. We excluded migrated code partly manual and partly automatic. We realized automatic exclusion, among others, by no longer matching on a function's name only, but by including the type of its first argument as well.

6. Threats to validity

This section discusses the validity of our study based on three types of threats: construct, internal, and external validity.

Threats to construct validity come from the way in which we evaluated our case study. Our study focused on the usefulness of our overviews, but we did not compare them with overviews produced using other approaches. We also did not investigate whether a bias might be present in our overviews. Yet, we are aware that the overviews are based on the latest version of the codebase, which does not cover the design space homogeneously but only reflects the currently realized solution.

Threats to internal validity come from the way in which we carried out our case study. Our study focused on qualitative aspects and ignored quantitative aspects. The time to create the custom static analysis was not considered, although some stakeholders considered it a performance indicator. Finally, the method is intended for software engineers, yet in the case studies the researchers developed the custom static analysis.

Threats to external validity come from the way in which our results will generalize to other in-house libraries and codebases. We only presented three case studies from the Dutch high-tech industry. The case studies were not randomly chosen, yet they vary in terms of the involved company, programming language, and type of in-house library.

7. Related work

We have organized the related work section into two subsections. The subsections relate our work to the fields of knowledge extraction and retrieval and of static analysis.

7.1. Knowledge extraction and retrieval

Different methods exist within the field of knowledge extraction and retrieval, including reverse engineering, program comprehension, and mining software repositories.

Tonella et al. (2007) define reverse engineering as follows: "Every method aimed at recovering knowledge about an existing software system in support to the execution of a software engineering task". Tonella et al. (2007) emphasize that no hidden or lost structure is recovered: "Rather, a structure is superimposed in order to facilitate the execution of some software engineering task". Raibulet et al. (2017) focus on model-driven reverse engineering and distinguish among others the following features: Automation level, scope, type of analysis, and availability of case studies. Furthermore, they identified the following main model-driven reverse engineering concepts:

- discovery: obtain models from analyzed systems;
- transformation: modification and abstraction of models to obtain the target representation of the analyzed systems; and
- metamodels: drivers for the discovery and transformation activities.

The method fits the reverse engineering definition: It superimposes a structure based on custom code patterns to create overviews. The method is semi-automated, has a custom specific scope, uses static analysis, and has been applied in three case studies. It is however not a model-driven approach, since it lacks explicit models, metamodels, and transformations.

Program comprehension is an important human factor in software engineering (Siegmund and Schumann, 2015). Human comprehension of a codebase correlates with that codebase's parameters such as content, language, layout, and size (Siegmund and Schumann, 2015). Humans apply several models for program comprehension, including top-down and bottom-up models (Siegmund and Schumann, 2015).

To enhance insight, our custom static analysis should also 'comprehend' the codebase. On the one hand, we expect that the comprehension of our custom static analysis of the codebase will also correlate with that codebase's parameters. However, we expect that the correlation will be different compared to human analysis, due to, among others, differences in memory and in sensitivity to, e.g., layout and regularity. The method also uses multiple 'models' for program comprehension with a local and global scope.

Approaches for mining software repositories analyze artifacts related to software development, such as source code version-control systems, requirements- and bug-tracking systems, communication archives, and deployment logs (Kagdi et al., 2007; de Freitas Farias et al., 2016). The artifacts related to software development do not cover the design space homogeneously but strongly focus on the solution evolving over time. The analysis techniques include metadata analysis, source code static analysis, source code differencing, clone detection, and classification supervised learning (Kagdi et al., 2007). The field of mining software repositories has a bias towards comprehension of software changes and evolution (Kagdi et al., 2007; de Freitas Farias et al., 2016).

Our custom static analysis statically analyzes only one version of the codebase, and aims at enhanced insight related to only a part of the codebase: the usage of an in-house library.

7.2. Static analysis

Many analysis tools "are unfocused, and take place in the absence of any indication of which properties are of interest to the consumer of the analysis information" (Jackson and Rinard, 2000). These analysis tools have generic goals and aim to find weaknesses and code smells, see, e.g., TiCS,[15] Coverity (Bessey et al., 2010), and HLint[16]; to generate related artifacts, such as skeletons, stubs, and documentation, see, e.g., Doxygen[17]; or to measure and quantify, see, e.g., GNATmetric.[18]

Custom static analysis

Some static analysis tools, like GNATcheck and SonarQube, support custom rules, but these are rarely used (Beller et al., 2016; Christakis and Bird, 2016). Furthermore, many analyses need more expressiveness than is provided, as, e.g., reported by Mendonça et al. (2018). Finally, these analysis tools favor overviews that are file-location oriented, which complicates analyses that need to produce other kinds of overviews.

Some static analysis tools, like Eclipse CDT and Clang, expose their Abstract Syntax Tree (AST). Although these tools enable all possible custom analyses, they require to learn a complex, abstract syntax tree that is typically not designed for analysis purposes only. Mendonça and Kalinowski (2022) even concluded that expecting developers which are not rule experts to develop custom "rules directly using the AST is unrealistic". The following alternatives are proposed to simplify custom static analysis:

- Domain Specific Languages, such as SmPL, that is based on the patch syntax (Padioleau et al., 2006), and Source Code Pattern Language, that uses naming conventions and markups embedded in the concrete syntax (Silva and Mendonça, 2021),
- domain-specific libraries, such as Clang-tidy[19] that provide a programming interface that does not hide but only aims to simplify working with the AST, and
- domain-specific libraries, such as Renaissance-Ada, that provide a programming interface by providing functions to find, filter, and replace code fragments using patterns expressed in (an extended version of) the concrete syntax, while hiding the complexities associated with matching abstract syntax trees as much as possible.

[15] https://www.tiobe.com/products/tics.
[16] https://github.com/ndmitchell/hlint.
[17] https://www.doxygen.nl.
[18] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html#the-gnat-metrics-tool-gnatmetric.
[19] https://clang.llvm.org/extra/clang-tidy.

Methods to create custom static analyses

Meta-level compilation (Engler et al., 2000) is a method to find violations of custom, system-specific rules. Instead of having all software engineers to obey rules as well as they can, despite approximate understanding of these rules and the system's complexity, meta-level compilation allows for each rule that a single software engineer, who understands that rule, writes a custom compiler extensions to automatically check that rule on everyone's code. The custom compiler extensions are written in a domain-specific language that supports patterns and state machines. Code is checked by searching for these patterns and, when matched, triggering transitions between states in a state machine. The custom compiler extensions find bugs, but do not guarantee their absence. The custom compiler extensions might produce false positives. These false positives can be reduced significantly by adding some amount of global analysis or system-specific knowledge.

Pattern-driven maintenance (Mendonça et al., 2018; Mendonça and Kalinowski, 2022) is a method to create custom static analysis rules for defect localization. Pattern-driven maintenance provides support to identify and document patterns from failure sources, such as error logs and issue reports; to implement rules to find instances of these patterns; and to include context analysis to reduce the number of false positives and make the rule acceptable for inclusion in the development process. The last aspect – improvement of analysis precision – has, according to Muske and Serebrenik (2016), "been extensively considered in the literature". Pattern-driven maintenance requires that the initial pattern should be broad enough to find all instances but might include false positives (Mendonça et al., 2018), but does not provide any support to check this requirement. The static analyses developed using pattern-driven maintenance range from general, i.e., application architecture and coding style specific, to custom, i.e., application-specific (Mendonça and Kalinowski, 2022).

The proposed method advocates custom static analysis that is specific to a given codebase to be cost-effective and to achieve the sweet spot in information abstraction: providing domain-specific insight without overwhelming details. Reusing standard analysis techniques, like call graphs, inheritance trees, and data flow analysis, makes developing the static analysis cost-effective, yet some customization is considered essential to achieve the sweet spot. Similarly, explicit checking for relevant code fragments that do not match any pattern is considered necessary to ensure the validity and completeness of the static analysis.

8. Discussion

We made the following general observations in our case studies:

- Despite the many differences between the three case studies, such as desired insights, codebases, and organizations, the method was applicable in all case studies.
- Using the method, we could develop a custom static analysis in all case studies.
- We needed customization in all case studies to enhance insight into the usage of the in-house libraries.
- The combination of domain knowledge with generic structures was crucial to get valuable insights in all case studies.
- Handling design patterns for decoupling was the biggest challenge and the main source for the complexity of the custom static analysis in all case studies.
- The industrial codebases were more regular than expected, possibly thanks to coding standards and guidelines, copy-paste-modify habits, the use of linters, and software engineers who know they have to maintain the code.

The method was developed, evolved, and validated in three exploratory case studies. We consider the following steps the most relevant ones in the evolution of the method:

1. During the "Blackboard" case, we observed the feasibility and usefulness of creating overviews of a specific in-house libraries using custom static analysis. Furthermore, we realized that all artifacts in the codebase might be relevant, and cross-checks are needed to ensure the validity and completeness of the custom overviews.
2. During the "Logging" case, we introduced additional cross-checks based on event logs from tests. We refined the discussion on what "the method" is, and we enriched our method with evaluation of expressions at compile-time via data flow analysis. In this case we also benefited from the reduction of variation in the codebase that kept the custom static analysis simple.
3. During the "Inspection" case, we realized the need to split the analysis into two parts. Dealing with (extreme) decoupling, we needed to retrieve the global context from multiple earlier nested method calls.

The effort to develop custom static analysis is considerable. Yet for large codebases, that effort is already smaller than a single manual analysis, even when using find functionality based on text and regular expressions. But also for smaller codebases, the effort might still be smaller, not only when the same analysis must be performed multiple times on variants of the codebase but also when iterations of the analysis are needed to achieve the desired quality. In manual and custom static analyses, iterations are needed when assumptions made in the analysis turn out to be incorrect, and when opportunities for improvements are discovered, such as additional information that can be extracted from the codebase and more in-depth analysis that can be performed.

We started this research despite the risk that the industrial codebases were so irregular with respect to domain-specific information that the development of a custom static analysis would not be effective and efficient. Our experience in these case studies has lowered our estimate of that risk and is inline with an observation of Hindle et al. (2016): "programming languages, in theory, are complex, flexible and powerful, but, "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive".

An explanation may be that engineering teams have processes, such as mandatory code reviews with check lists, and use tools, such as linters, to improve code regularity. In addition, some programming languages, such as Python, have design principles like "There should be one – and preferably only one – obvious way to do it"[20] that stimulate code regularity. Other programming languages, such as Perl (Wall et al., 2000), have slogans like "There's More Than One Way to Do It". Yet, even Perl's creator, Larry Wall, hesitates to make ten ways to do something.

Many software engineers adhere to the proverb "if it ain't broke, don't fix it". Yet, when do code owners consider a codebase broken? In the case studies, multiple fixes were proposed that removed code irregularities from the codebase to keep the custom static analysis simple. Of course, these fixes not only improved analyzability but also other properties, such as simplicity and readability. All proposed fixes were accepted by the code owners. In other words, we observed that code owners considered their codebase broken not only when exposing incorrect behavior but also when containing code irregularities.

The proposed method needs tools for static analysis and visualization. Some programming languages currently lack static analysis tools. Enhancing insight into the usage of in-house libraries written in those programming languages will be quite a challenge as also these tools have to be developed.

Static analysis tools can be classified in two categories: targeting either a specific programming language or all programming languages. Tools of the latter category still need a programming language specific adapter to work correctly. At the time of our case studies no tool in the latter category supported the languages we needed. However, currently Rascal[21] has added support for C/C++ and Ada and has become a viable alternative, as is exemplified in Schuts et al. (2022). For the case studies we selected tools from the first category: Eclipse CDT for the C++ programming language and Libadalang for the Ada programming language. However, Clang and ASIS are equally valid alternatives for these respective languages.

9. Conclusions

In this paper, we have presented a method to enhance insight into the usage of in-house libraries. This method was applied in three exploratory case studies involving industrial C++ and Ada codebases with in-house libraries. We draw the following conclusions:

- We could develop a custom static analysis to enhance insight for all industrial codebases of the case studies. The development turned out to be easier than initially envisioned, since the industrial codebases were more regular than expected.
- The enhanced insight immediately revealed a lot of quick-wins, confirming that you can only manage what you can measure. Many of the quick-wins, such as the removal of domain-specific redundant code, are directly linked to exploiting domain knowledge.
- The custom static analysis had a set of code patterns for proper library usage, and included cross-checks to detect deviating library usage. These cross-checks were crucial for the validity and completeness of the analysis.
- The automation of the custom static analysis improved the quality of the analysis results and reduced the time needed for analysis considerably.

10. Future work

We would like to apply the proposed method also to open-source codebases: Not only to verify that the method generalizes beyond industrial codebases, but also to provide examples that are accessible as additional documentation and for further research. So we invite open-source communities to contact us whenever they want to enhance insight into the usage of in-house libraries within their codebase.

Industrial codebases contain more domain-specific information besides the usage of in-house libraries. We would like to extend the proposed method to include this information as well. Hence, we would like to generalize the method to "enhance insight into the usage of in-house libraries" to "enhance domain-specific insight into an industrial codebase". We consider a generic way to handle decoupling patterns crucial for this generalization.

We observed in the case studies that the enhanced insight, e.g., the detection of no longer used code, triggers changes in the codebase. These changes are often not complicated, e.g., the recording of the obsolete identifiers to avoid incidental reuse and the removal of no longer used code. So we would like to incorporate automatic changes in the method in the future.

CRediT authorship contribution statement

Piërre van de Laar: Methodology, Conceptualization, Investigation, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Rosilde Corvino:** Methodology, Conceptualization, Investigation, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Arjan J. Mooij:** Methodology, Conceptualization, Investigation, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Hans van Wezep:** Resources, Validation. **Raymond Rosmalen:** Resources, Validation.

[20] https://www.python.org/dev/peps/pep-0020.

[21] https://www.rascal-mpl.org.

Declaration of competing interest

We would like to declare the following potential conflicts of interest that may be perceived as influencing the research reported in our article.

Hans van Wezep and Raymond Rosmalen are employed by Philips and ITEC, respectively, which develop and maintain the in-house libraries that were analyzed in the study. While the research was conducted independently of any commercial interests, the results have had implications for the further development and use of these libraries.

In addition, Pierre van de Laar, Rosilde Corvino, and Arjan J. Mooij acknowledge that some of the tools and techniques used in the study are developed by TNO ESI as part of the study, and we have an interest in their continued use and development.

We believe that these conflicts of interest do not compromise the validity or objectivity of our research findings. However, we wanted to disclose these relationships in the interest of transparency and to allow the editors and reviewers of the Journal of Systems and Software to make an informed decision about the publication of our article.

We confirm that this Declaration of Interest Statement is accurate and complete to the best of our knowledge.

Data availability

The data that has been used is confidential.

Acknowledgments

The research was carried out as part of the Bright and Vivace programs under the responsibility of TNO-ESI with ITEC and Royal Philips as the carrying industrial partners. The Bright and Vivace programs are supported by the Netherlands Organisation for Applied Scientific Research TNO. The authors like to thank Jeroen Ketema for his contributions to these programs and his valuable input to the discussions that eventually led to this article.

References

Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M., 2019. A systematic evaluation of static API-misuse detectors. IEEE Trans. Softw. Eng. 45 (12), 1170–1188. http://dx.doi.org/10.1109/TSE.2018.2827384.

Beller, M., Bholanath, R., McIntosh, S., Zaidman, A., 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1. IEEE Computer Society, pp. 470–481. http://dx.doi.org/10.1109/SANER.2016.105.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., Engler, D.R., 2010. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53 (2), 66–75. http://dx.doi.org/10.1145/1646353.1646374.

Christakis, M., Bird, C., 2016. What developers want and need from program analysis: an empirical study. In: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. ACM, pp. 332–343. http://dx.doi.org/10.1145/2970276.2970347.

Dams, D., Ketema, J., Kramer, P., Mooij, A.J., Rădulescu, A., 2021. Developing and applying custom static analysis tools for industrial multi-language code bases. In: Catolino, G., Nucci, D.D., Tamburri, D.A. (Eds.), Proceedings of the 20th Belgium-Netherlands Software Evolution Workshop, Virtual Event / 's-Hertogenbosch, The Netherlands, December 7-8, 2021. In: CEUR Workshop Proceedings, vol. 3071, CEUR-WS.org, URL: http://ceur-ws.org/Vol-3071/paper17.pdf.

de Freitas Farias, M.A., Novais, R.L., Júnior, M.C., da Silva Carvalho, L.P., Mendonça, M.G., Spínola, R.O., 2016. A systematic mapping study on mining software repositories. In: Ossowski, S. (Ed.), Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. ACM, pp. 1472–1479. http://dx.doi.org/10.1145/2851613.2851786.

Engler, D.R., Chelf, B., Chou, A., Hallem, S., 2000. Checking system rules using system-specific, programmer-written compiler extensions. In: Jones, M.B., Kaashoek, M.F. (Eds.), 4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000. USENIX Association, pp. 1–16.

Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131. http://dx.doi.org/10.1145/2902362.

Horváth, G., Szécsi, P., Gera, Z., Krupp, D., Pataki, N., 2018. [Engineering paper] challenges of implementing cross translation unit analysis in clang static analyzer. In: 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018. IEEE Computer Society, pp. 171–176. http://dx.doi.org/10.1109/SCAM.2018.00027.

Jackson, D., Rinard, M.C., 2000. Software analysis: a roadmap. In: Finkelstein, A. (Ed.), 22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000. ACM, pp. 133–145. http://dx.doi.org/10.1145/336512.336545.

Jbara, A., Feitelson, D.G., 2014. On the effect of code regularity on comprehension. In: Roy, C.K., Begel, A., Moonen, L. (Eds.), 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014. ACM, pp. 189–200. http://dx.doi.org/10.1145/2597008.2597140.

Kagdi, H.H., Collard, M.L., Maletic, J.I., 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maint. Res. Pract. 19 (2), 77–131. http://dx.doi.org/10.1002/smr.344.

Klusener, S., Mooij, A.J., Ketema, J., van Wezep, H., 2018. Reducing code duplication by identifying fresh domain abstractions. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018. IEEE Computer Society, pp. 569–578. http://dx.doi.org/10.1109/ICSME.2018.00020.

van de Laar, P., Mooij, A., 2022. Renaissance-Ada: tools for analysis and transformation of Ada code. Ada User J. 43 (3), 165–170, URL: https://www.ada-europe.org/archive/auj/auj-43-3-withcovers.pdf.

Mendonça, D.S., Kalinowski, M., 2022. An empirical investigation on the challenges of creating custom static analysis rules for defect localization. Softw. Qual. J. 30 (3), 781–808. http://dx.doi.org/10.1007/S11219-021-09580-Z.

Mendonça, D.S., da Silva, T.G., de Oliveira, D.F., Brandão, J.S., Lopes, H., Barbosa, S.D.J., Kalinowski, M., von Staa, A., 2018. Applying pattern-driven maintenance: a method to prevent latent unhandled exceptions in web applications. In: Oivo, M., Fernández, D.M., Mockus, A. (Eds.), Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018. ACM, pp. 31:1–31:10. http://dx.doi.org/10.1145/3239235.3268924.

Mooij, A.J., Ketema, J., Klusener, S., Schuts, M., 2020. Reducing code complexity through code refactoring and model-based rejuvenation. In: Kontogiannis, K., Khomh, F., Chatzigeorgiou, A., Fokaefs, M., Zhou, M. (Eds.), 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020. IEEE, pp. 617–621. http://dx.doi.org/10.1109/SANER48275.2020.9054823.

Mossienko, M., 2006. Structural search and replace: What, why, and how-to. URL: https://www.jetbrains.com/idea/docs/ssr.pdf.

Muske, T., Serebrenik, A., 2016. Survey of approaches for handling static analysis alarms. In: 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016. IEEE Computer Society, pp. 157–166. http://dx.doi.org/10.1109/SCAM.2016.25.

Ossendrijver, R., Schroevers, S., Grelck, C., 2022. Towards automated library migrations with error prone and refaster. In: Hong, J., Bures, M., Park, J.W., Cerný, T. (Eds.), SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022. ACM, pp. 1598–1606. http://dx.doi.org/10.1145/3477314.3507153.

Padioleau, Y., Hansen, R.R., Lawall, J.L., Muller, G., 2006. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In: Probst, C.W. (Ed.), Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, PLOS 2006, San Jose, California, USA, October 22, 2006. ACM, p. 10. http://dx.doi.org/10.1145/1215995.1216005.

Potts, C., 1993. Software-engineering research revisited. IEEE Softw. 10 (5), 19–28. http://dx.doi.org/10.1109/52.232392.

Raibulet, C., Fontana, F.A., Zanoni, M., 2017. Model-driven reverse engineering approaches: A systematic literature review. IEEE Access 5, 14516–14542. http://dx.doi.org/10.1109/ACCESS.2017.2733518.

Reiss, S.P., 2005. The paradox of software visualization. In: Ducasse, S., Lanza, M., Marcus, A., Maletic, J.I., Storey, M.D. (Eds.), Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, September 25, 2005. IEEE Computer Society, pp. 59–63. http://dx.doi.org/10.1109/VISSOF.2005.1684306.

Rogerson, D., 1997. Inside COM: Microsoft's Component Object Model. Microsoft Press.

Schröter, I., Krüger, J., Siegmund, J., Leich, T., 2017. Comprehending studies on program comprehension. In: Scanniello, G., Lo, D., Serebrenik, A. (Eds.), Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017. IEEE Computer Society, pp. 308–311. http://dx.doi.org/10.1109/ICPC.2017.9, URL: https://ieeexplore.ieee.org/xpl/conhome/7959731/proceeding.

Schuts, M.T.W., Aarssen, R.T.A., Tielemans, P.M., Vinju, J.J., 2022. Large-scale semi-automated migration of legacy C/C++ test code. J. Softw.: Pract. Exp. 52 (7), 1543–1580. http://dx.doi.org/10.1002/spe.3082.

Siegmund, J., Schumann, J., 2015. Confounding parameters on program comprehension: a literature survey. Empir. Softw. Eng. 20 (4), 1159–1192. http://dx.doi.org/10.1007/s10664-014-9318-8.

Silva, D., Mendonça, D.S., 2021. SCPL: A markup language for source code patterns localization. In: Vasconcellos, C.D., Roggia, K.G., Collere, V., Bousfield, P. (Eds.), 35th Brazilian Symposium on Software Engineering, SBES 2021, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021. ACM, pp. 127–132. http://dx.doi.org/10.1145/3474624.3476017.

Tonella, P., Torchiano, M., Bois, B.D., Systä, T., 2007. Empirical studies in reverse engineering: state of the art and future trends. Empir. Softw. Eng. 12 (5), 551–571. http://dx.doi.org/10.1007/s10664-007-9037-5.

Wall, L., Christiansen, T., Orwant, J., 2000. Programming Perl - There's More Than One Way to Do It, third ed. OReily Media.

Wlodarski, L., Pereira, B., Povazan, I., Fabry, J., Zaytsev, V., 2019. Qualify first! A large scale modernisation report. In: Wang, X., Lo, D., Shihab, E. (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019. IEEE, pp. 569–573. http://dx.doi.org/10.1109/SANER.2019.8668006.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2018. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976. http://dx.doi.org/10.1109/TSE.2017.2734091.

Piërre van de Laar studied theoretical and computational physics and received his Ph.D. at the Catholic University of Nijmegen, The Netherlands. He worked at Philips Research in Eindhoven, where he investigated the exploitation of architecture description languages for product families, visualization, verification, aspect-orientation, and dependability. He is currently employed by TNO-ESI. He participated in multiple public–private research projects, cooperating with Philips, Thales, ASML, VDL, ITEC, and most Dutch universities. He focuses on evolvability of embedded systems, situational awareness, model based testing, rejuvenation, and synthesis-based engineering.

Rosilde Corvino is an IT professional with experience in embedded software development and project management. She holds a Master of Science in Electronic Engineering from the Politecnico di Torino and a Ph.D. in micro and nanoelectronics from the Universite Joseph Fourier of Grenoble. Throughout her career, Rosilde has held technical positions in processor design, compiler design, and software development, working for organizations such as INPG and INRIA in France and TU/e, Intel, and TNO-ESI in The Netherlands. At TNO-ESI, she leads research activities to define effective and efficient approaches to dealing with software legacy.

Arjan Mooij works as senior lecturer for software engineering at the Zurich University of Applied Sciences (ZHAW) in Switzerland. He holds a M.Sc. and Ph.D. degree in Computer Science from Eindhoven University of Technology in The Netherlands. Before joining ZHAW, he worked over 12 years as research fellow for TNO-ESI in The Netherlands, where he pioneered the research line on software legacy, including both software analysis and software transformation. This work includes the development of methods and tools based on real case studies with the industry. The results are available in publications, learning tracks, and commercial services.

Hans van Wezep is a seasoned software architect with over 20 years of experience in various domains, but by far the most in the medical domain at Philips. He has been involved in a multitude of development projects bringing state of the art X-ray scanners to the market. For over 10 years, he has been involved in various research projects regarding state-of-the-art model-based software design methods, and model-based software transformations and rejuvenations methods and integrating them in an industrial setting in collaboration with TNO. He graduated with a Computer Science degree from the University of Twente.

Raymond Rosmalen, More than 25 years of experience in the field of machine vision, and currently the Machine Vision Technology Architect within the Innovation Team at ITEC. Developing backend semiconductor assembly equipment and specialized in integrated and standalone machine vision solutions. Focusing on breakthrough vision technologies for the next generation of highly efficient quality inspections at the lowest cost of ownership. Studied Physics at Radboud University in Nijmegen and holds a Ph.D. on high energy physics through contributions to the LEP experiment at CERN.