UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea Triennale in Informatica

Tesi di Laurea

INTEGRATING ECLAIR STATIC ANALYSIS IN IDES USING THE LANGUAGE SERVER PROTOCOL

Integrazione dell'analizzatore statico ECLAIR in IDE tramite il Language Server Protocol

Candidato:
Nicolò Fuccella

Relatore:
Prof. Roberto Bagnara

Anno Accademico 2021-2022

Abstract

ECLAIR is a powerful platform for software verification with a strong focus on the development of high-integrity systems, including safety-and security-critical systems. ECLAIR works on the desktop and on the server to analyze entire projects to find possible defects. Following the "shift-left" trend (that is, performing verification activities starting from the early development phases), the work described in this thesis aims at providing a proof-of-concept of "immediate feedback software verification" using ECLAIR. This entails giving feedback to developers about their code correctness while they're writing the code, directly in the IDE they use for development and without slowing down their work. One of the challenges to be faced is the compatibility with multiple IDEs and editors. The proliferation of such coding environments, each one with its own extension standards and practices, urges a decoupling between the IDE's extension/plugin interface and the verification-specific smarts. The Language Server Protocol aims precisely at that: having a standardized protocol for communications between a Language Server and the IDE, a single Language Server can be reused for multiple development environments. The transition from the traditional way in which static analysis results are consumed to this "immediate feedback" modality presents another challenge: analysis time becomes critical and a fully satisfactory solution requires sophistication in the static analysis techniques (such as incrementality and parallelism). We developed a proof-of-concept implementation of the integration of ECLAIR with IDEs using the Language Server Protocol. This thesis describes the underlying ideas, the challenges, the possible solutions explored during the implementation of the proof-of-concept, the lessons learned, and the further developments that, in our opinion, have the potential of turning the prototype into a fully fledged software development tool.

Sommario

ECLAIR è un potente strumento per la verifica del software focalizzato sullo sviluppo di sistemi high-integrity, safety-critical e security-critical. ECLAIR è pensato per funzionare sia sul desktop che sul server per analizzare interi progetti al fine di rilevarne eventuali difetti. Seguendo la tendenza verso il cosiddetto "shift-left" (che consiste nell'anticipare le attività di verifica già dalle prime fasi dello sviluppo), il progetto descritto in questa tesi ha l'obiettivo di fornire una prova di fattibilità di un "sistema di verifica con feedback immediato" usando ECLAIR. Questo al fine di dare feedback agli sviluppatori in merito alla correttezza del loro codice durante la stesura del programma, direttamente dall'IDE che già utilizzano per lo sviluppo e senza subire rallentamenti. Una delle sfide da affrontare in un progetto di questo tipo è garantire la compatibilità con i vari IDE ed editor. La proliferazione di questi ambienti di sviluppo, ciascuno con i suoi standard e approcci consigliati per la realizzazione di estensioni, rende necessaria una separazione tra l'interfaccia dell'estensione/plugin dell'IDE e le funzionalità prettamente legate all'analisi. Il Language Server Protocol si pone come obiettivo precisamente questo: definendo un protocollo standardizzato per le comunicazioni tra un Language Server e l'ambiente di sviluppo, un solo Language Server può essere utilizzato per vari ambienti di sviluppo senza apportare modifiche. Il passaggio dal tradizionale approccio alla fruizione dei risultati dell'analisi statica a questa modalità "con feedback immediato" presenta altre sfide: il tempo di analisi diventa un fattore determinante e la realizzazione di una soluzione richiede l'impiego di alcune tecniche di analisi piuttosto sofisticate (per esempio incrementalità e parallelismo). Abbiamo realizzato un'implementazione che costituisce una prova di fattibilità dell'integrazione di ECLAIR in alcuni ambienti di sviluppo usando il Language Server Protocol. Questa tesi ha l'obiettivo di descrivere le idee sottostanti, le sfide affrontate, le possibili soluzioni esplorate durante l'implementazione del prototipo, gli insegnamenti appresi, e i futuri sviluppi che, a nostro avviso, hanno il potenziale di trasformare il prototipo in uno strumento di sviluppo software a pieno titolo.

Contents

List of Figures

Introduction

Nowadays static analysis is more and more common in every development scenario. In some cases, especially safety-critical ones, it is even enforced by regulation and industry standards. Although it can improve code quality dramatically, developers are often reluctant to implement it. This is mainly due to the fact that most of these tools are designed to run separately from development environments: they return extensive lists of notifications which must be inspected one by one in a time-consuming fashion. Analyzers manufacturers know the situation and tried different approaches: it is now common for them to build on top of their existing tools integrations with IDEs and CI (Continuous Integration) pipelines. The core of the analyzer and its code remains pretty much the same, what changes is the interface that the end users can rely on. This thesis focused on the integration possibilities between the IDE and the analyzer, in order to assist the user during the coding. This can be achieved in different ways: from a simple button in the IDE that triggers the analysis to more integrated experiences.

In a publication by Yuriy Tymchuk, Mohammad Ghafari and Oscar Nierstrasz in occasion of the 2018 IEEE/ACM 26th International Conference we can read:

> *"We learned that the availability of our tool as a default IDE feature and its automatic execution are the main reasons for its adoption. Moreover, the fact that immediate feedback is provided directly in the related development context is essential to keeping developers satisfied, although in certain cases feedback delivered later was deemed more useful." [13]*

This confirmed our hunch, and we wanted to try to build an IDE extension on top of an existing analyzer, ECLAIR by BUGSENG[1], to better understand the challenges and the state of the art of this kind of integration.

[1] https://www.bugseng.com/eclair

BUGSENG had already explored integrations with CI pipelines and the analyzer was already capable of exporting reports in different formats: given this background we could fully concentrate on the IDE side. After a brief analysis of other tools and developer feedback, we outlined the experience we wanted to present to the users: they should still be able to trigger an analysis manually, but in addition to that whenever a file was changed or opened for the first time, it should run automatically, so that we could always present the most updated warnings to the user. However, each IDE has its own primitives and APIs to be called in order to integrate the functionalities we wanted. It would have been time consuming and a maintenance nightmare to develop extensions for all the mostly used IDEs, until we learned about the Language Server Protocol[2]: defining a common communication standard between the IDE and a decoupled Language Server, the protocol handles the heavy lifting of reducing all the IDEs to a common interface from the server point of view. This way we could integrate the analysis smarts in the Language Server, who could seamlessly deliver the warnings to the IDE, and fully concentrate on problems like incrementality and parallelism.

After some initial experiments, we built a prototype of a VSCode extension which relies on the Language Server Protocol for all the communication between a Language Server and the IDE. The Language Server behaves as an orchestrator between two tools: the ECLAIR analyzer itself and *eclair_report*, which serves the violations met.

We believe that tools like this, which entails performing the analysis during the software development phase, can give an important boost to developers and allow them to conciliate speed and code quality assuring that bugs, inconsistencies, design issues and other quirks won't reach the production environment, but will be dealt with before.

[2]https://microsoft.github.io/language-server-protocol

Part I

Preliminary Notions

This part of the thesis introduces the main preliminary notions and premises that are needed for the presentation of the contents of Part 2, and that provide the reader with the necessary context information.

In Chapter 1, the current approaches at verifying the correctness of programs are presented. After an introduction about testing in general, the reasons behind static analysis are presented and the pros and cons are weighed.

Chapter 2 will then focus on shift-left testing movement and what it actually implies.

Moving to Chapter 3, the ECLAIR static analyzer is presented, on top of which the proof-of-concept has been realized. Before talking about integrating ECLAIR static analysis in IDEs, it is important to understand how ECLAIR works and the reasoning behind it. Chapter 3 also includes an introduction to MISRA C and its guidelines.

Lastly, Chapter 4 presents the Language Server Protocol on top of which all the communications between the IDEs and the analysis system are performed. Thanks to this protocol, we were able to decouple the IDEs extensions from the analysis tools.

Chapter 1

Verifying the Correctness of Programs

Today's dominant practice in the software industry is to demonstrate the correctness of programs empirically. Some inputs are fed to the tested program and the correctness of the output is verified against expected ones. In some cases exhaustive testing is possible, but in large software it generally is not. Just focusing on a single function, for example, with one 32-bit parameter it can possibly be exhaustively tested. With more than one 32-bit parameters or one or more 64-bit parameters, exhaustive testing suddenly becomes impossible.

However, the only thing we can actually prove with such an approach is that the program is incorrect: a single instance of incorrect behavior suffices to spot a problem. Unfortunately, when there is not an observation of incorrect behavior, we cannot know whether the program is correct or we have just not tested it with an input that would trigger an error. Some more sophisticated testing tries to choose the inputs so that all, or at least the majority of the possible execution paths are examined, and to test modules or units (testing in the small) as well as the overall software behavior (testing in the large).

Regardless of how sophisticated the testing is, empirical methods do not actually prove that a program is correct. Adopting a completely different approach, static analysis aims at giving more insights about some properties of the program without the need for the execution, but by examining the code. An analyzer scans all the code in a project, understands its structure and helps to check for vulnerabilities and ensure that it adheres to industry standards. So static analysis is mostly useful when looking for:

- security vulnerabilities;

- programming errors;
- coding standard or syntax violations;
- performance issues.

Integrating static analysis in the development workflow improves code quality, thanks to automated tools which are less prone to human error, increase the likelihood of finding vulnerabilities and, paired with normal testing methods, allows for more depth into debugging code.

However, static analysis comes with some drawbacks such as false positives and sometimes cryptic violation reports. In addition, enforcing too strict coding rules may slow down the development for little benefit.

Nowadays static analysis is a mandatory step in high level software development, in particular for safety-critical systems and is often enforced by common regulations. Thus we have seen a proliferation of such tools, which are commonly paired with code review automations and continuous integration pipelines on platforms like GitHub[1] or Gitlab[2]. These integrations perform the analysis after the code has been written or at least when it has reached a stable state (usually when changes are pushed to the repository or a pull request is opened). Some of these tools are used also during the code writing to improve the quality and reduce the errors before shipping the changes and performing more in-depth checks. This last use case opens a series of new requirements and problems: first of all, performing the analysis in the shortest amount of time becomes of the utmost importance, and while it is doable for short files, sometimes the codebases to consider can be huge and with a number of levels of abstractions. The analysis tooling, in order to be effective, must become a shared habit across the team, thus giving space to additional headaches: teams work on different platforms (Linux, MacOS, Windows, etc...) and with different IDEs/editors (Intellij Idea[3], VSCode[4] and Emacs[5] just to name a few). Acknowledging that, the static analysis tools must be compatible with each of them, with an increase in the number of softwares to be developed and maintained by the producers.

[1] https://github.com
[2] https://gitlab.com
[3] https://www.jetbrains.com/idea
[4] https://code.visualstudio.com
[5] https://www.gnu.org/software/emacs

Chapter 2

Shift-left testing

In an article dated September 1st 2001 appeared in Dr. Dobb's Journal, Larry Smith wrote:

> *"Shift-left testing is how I refer to a better way of integrating the quality assurance (QA) and development parts of a software project. By linking these two functions at lower levels of management, you can expand your testing program while reducing manpower and equipment needs - sometimes by as much as an order of magnitude." [12]*

This article is considered the manifesto of the shift-left movement, which is about moving the testing phase earlier in the software development lifecycle, shifting-left.

To give some background to the reader, up until the late 1990s, a typical software development process was sequential: the design was followed by the development, while testing and deployment were the latest phases of the project lifetime. Placing the testing phase at a late stage, bug fixing was costly and time-consuming. Sometimes developers had to even fully redesign the application in order to make everything behave correctly. Kent Beck and Cynthia Andres wrote about this

> *"Here is the dilemma in software development: defects are expensive, but eliminating defects is also expensive. However, most defects end up costing more than it would have cost to prevent them." [8]*

Listening to Larry Smith's suggestion, more and more teams try to integrate the quality assurance from the earlier phases of the development. The test-driven development is an approach that tries to do exactly this, designing the tests even before beginning the coding part. In addition, many tools

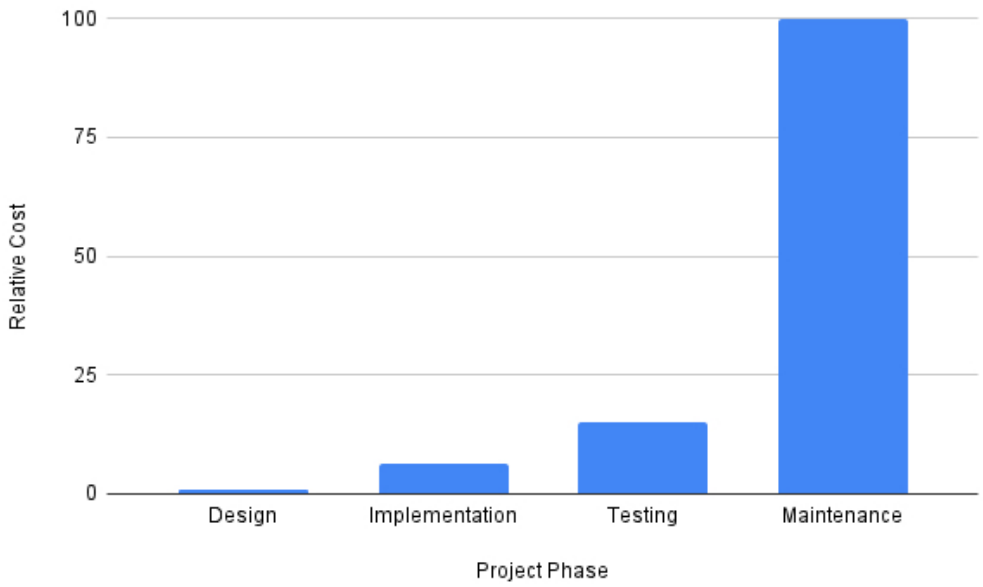


Figure 2.1: Relative Cost of Fixing Defects

were born to help the developers to "shift-left", like static analyzers.

It is important for the reader to understand that this kind of approach, aside from helping to find bugs early, thus reducing the costs implied by patches and code fixes, leads to a higher-quality product and codebase. This usually can help reduce the time to ship a product, avoiding unexpected issues and the need for refactoring existing parts of the code.

In a software industry that is everyday more and more bound to other fields, especially safety-critical ones like medical, automobilistic and aeroespacial just to name a few, software quality is of the utmost importance, hence the *"test early and often"* motto has become kind of a mantra for teams, which has now the instruments to catch defects as early as possible and in the least expensive way by integrating sane good practices and habits during the development.

Chapter 3

ECLAIR

ECLAIR is a powerful platform for software verification. At the moment of this writing the latest stable release, and the version adopted for the project described in Part II, is the 3.12.0. Applications range from coding rule validation, with a particular emphasis on the MISRA and BARR-C coding standards, to the computation of software metrics, to the checking of independence and freedom from interference among software components, to the automatic detection of important classes of software errors. ECLAIR is certified for use in safety-critical development ranging from automotive to aerospace use cases, moving to industrial and medical ones. It uses formal methods-based static code analysis techniques such as abstract interpretation and model checking combined with constraint satisfaction techniques to detect or prove the absence of certain run time errors in source code, and provides support for program analysis and verification, program test generation and program transformation.

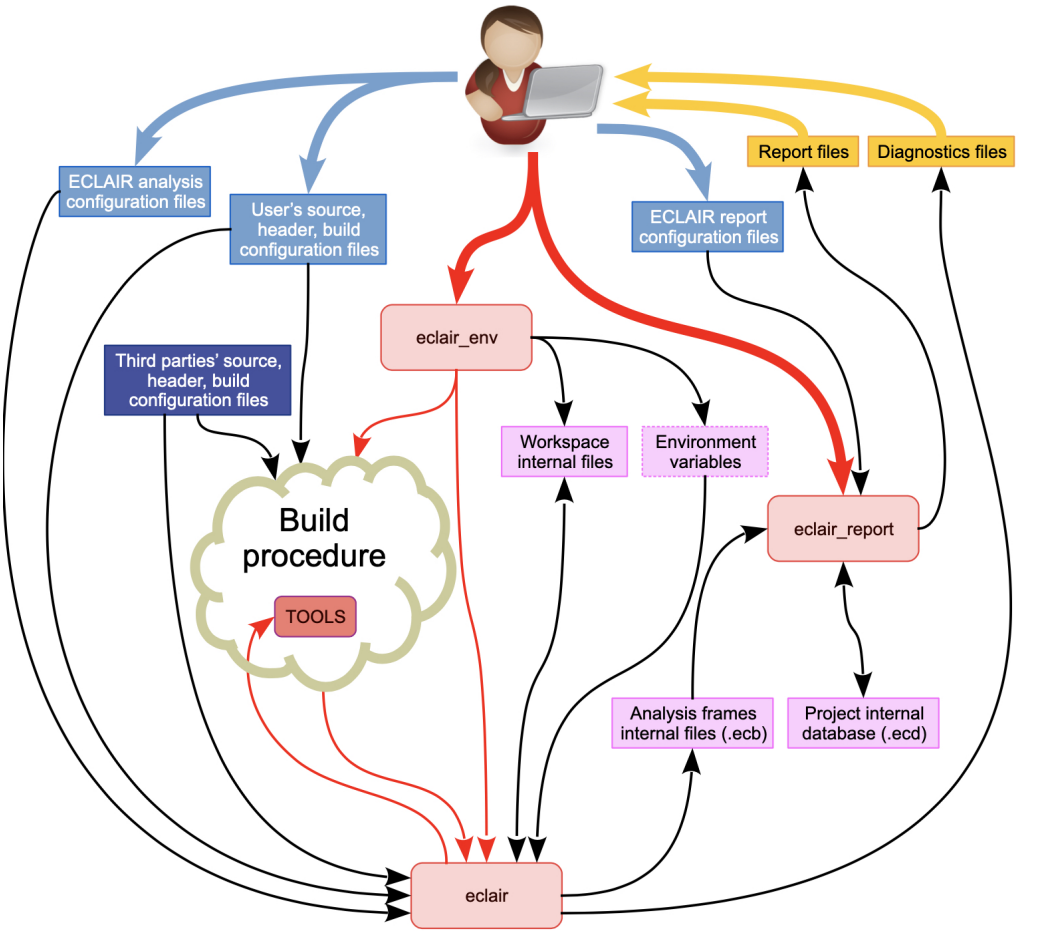


Figure 3.1: ECLAIR basic qualifiable architecture - Image copyright by BUGSENG srl, reproduced with permission.

3.1 Abstract interpretation

Abstract interpretation was formalized by the French computer scientists Patrick Cousot and Radhia Cousot in the late 1970s and it mainly consists of the automatic extraction of information about the possible execution paths of a program. The formalization of all these possible execution paths is called *concrete semantics*. The concrete semantics of a program is, in general, a non-computable, typically infinite mathematical object. Thus it is not possible to write a program to compute it. Having said that, all the non-trivial questions about the concrete semantics are undecidable.

However, we can consider a sound approximation of the concrete program semantics, called *abstract semantics*, and reason on it. Since the abstract

semantics covers all possible cases, we can use it to demonstrate safety properties of the program.

The abstract semantics is an artificial construct aimed at giving a computable approximation of the concrete semantics. It is important to pay attention to the "direction" of the approximation: it must "err on the safe side" in order to avoid false negatives, though at the expense of allowing false positives. However, once a good approximation is available, abstract interpretation can give precious insights into the program properties and inner workings.

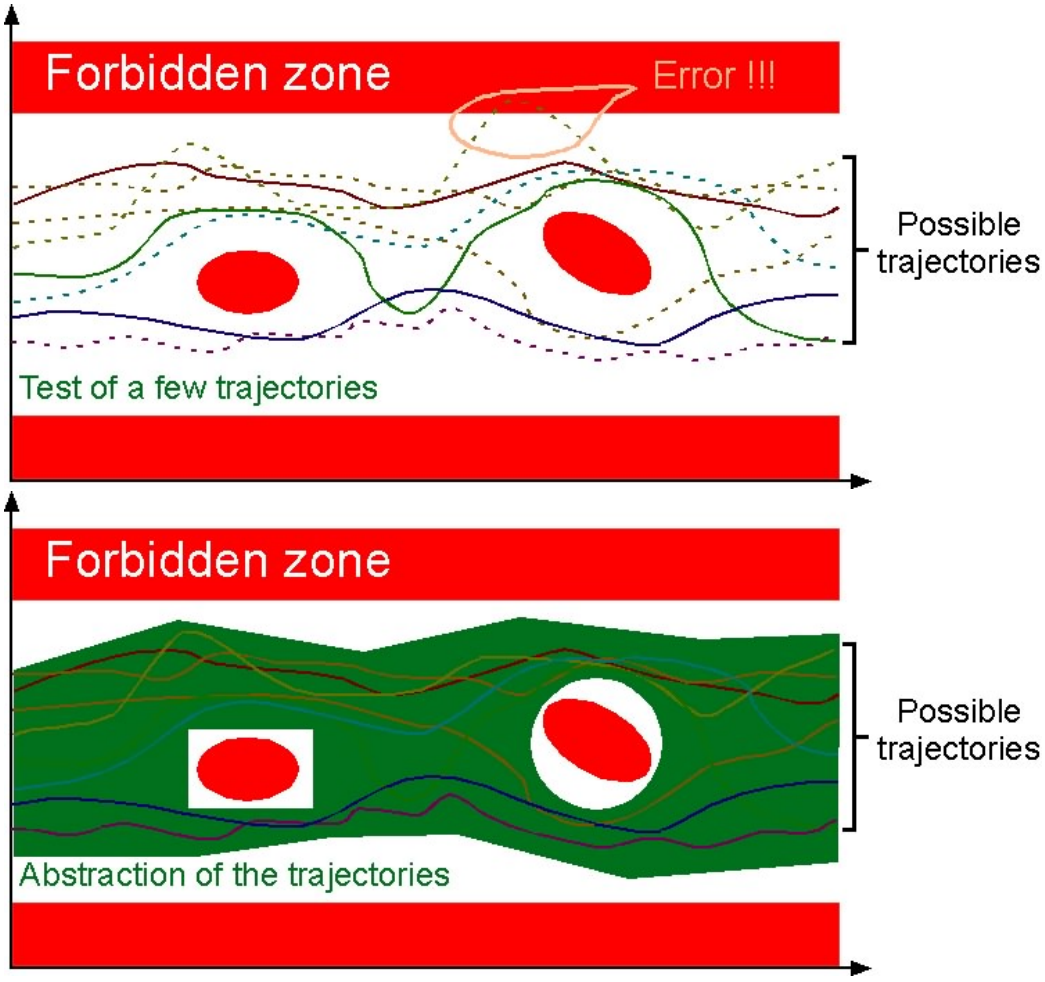


Figure 3.2: Abstract Interpretation in a Nutshell [9]

3.2 Model checking

Model checking is a method for checking whether a finite-state model of a system meets a given specification (also known as correctness). This verification can be performed by automatically searching the finite state space of the system looking to determine whether the property of the specification is met. To demonstrate that the specification is not met, the checker looks for counterexamples that can give precious insights about the reason for the failure. Having said that, the process of model checking can be summarized as follow:

- modeling, which consists of creating an abstract model of the system ignoring all the irrelevant details (eg. through Kripke structures);
- specification, which describes with a logical formalism (like temporal logic) properties that must be satisfied;
- verification, which checks that the properties are actually met by the system.

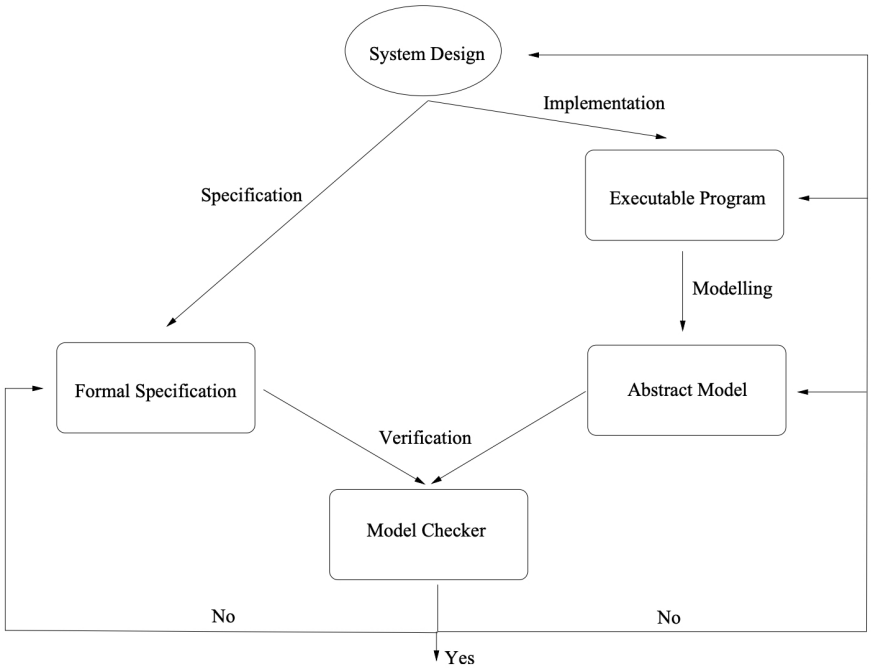


Figure 3.3: The process of model checking [10]

It is important to pay particular attention to the modeling and specification phases, since errors can make this method give false positives. Having said

that, model checking is a precious method used in several static analysis tools.

3.3 Constraint satisfaction

Constraint satisfaction is the process of finding a solution through a set of constraints that impose conditions some variables must satisfy. It is one of the subjects of investigation in the fields of artificial intelligence and operations research.

In particular, Constraint Satisfaction Problems (CSPs) form a specific class of search problems. A CSP is a triple which comprehend:

- variables, a finite and non empty set;
- domains, finite and non empty sets, one for each variable;
- constraints over variables, such that each constraint is a subset of the cartesian product of domains sets.

Solutions are combinations of values that can be assigned to the variables, from the respective domains, that satisfies all the constraints at the same time.

Constraint satisfaction has also been used to perform checks on programs during static analysis. Constraint propagation is an essential part of this method, which, relying on local consistency conditions, tries to reduce domains of variables, strengthen constraints, or create new ones. To give an idea about the possible use cases of such an approach let's consider an example found in "A Practical Approach to Verification of Floating-Point C/C++ Programs" by Bagnara et al. [7]. In Chapter 3 the expression

float phi_= asin(sin(dl) / cosh(ll));

is translated into static single assignment form and is decomposed. We can read about this:

> *"In this intermediate code representation, complex expressions are decomposed into sequences of assignment instructions where at most one operator is applied, and new variable names are introduced so that each variable is assigned to only once."*

The expression results in a list of statements like this:

```
1 float phi_; double z1, z2, z3, z4, z5, z6;
2 z1 = (double) dl;
3 z2 = sin(z1);
4 z3 = (double) ll;
5 z4 = cosh(z3);
6 z5 = z2/z4;
7 z6 = asin(z5);
8 phi_ = (float) z6;
```

and then, regarding line 6:

> *"Let us see how this can be used for program verification. As a first example, let us consider the question of whether the division z5 = z2 / z4 can give rise to a division by zero. Assume that all of the intervals are initially full - for instance, they contain all possible numerical floating-point values and all propagators are ready to run. We modify the interval associated to z4 to [-0, +0] and start propagation. At some stage, the indirect propagator for cosh will be called to possibly refine the interval for z3 starting from the interval of z4: a propagator correctly capturing a passable implementation of cosh will refine the label of z3 to the empty interval, thus proving that division by zero is indeed not possible."*

3.4 MISRA C

MISRA C is a set of software development guidelines and the de facto standard for developing software in C where safety, security and code quality are important. It was originally developed to fulfill the need for a "restricted subset of a standardized programming language" identified in the 1994 "Development guidelines for vehicle based software" and against the background of the emerging use of C for developing embedded software in automotive applications. Although originally specifically targeted at the automotive industry, it has evolved as a widely accepted model for best practices by leading developers in sectors including automotive, aerospace, telecom, medical devices, defense, railway, and others. MISRA standards are used to ensure that the code is:

- safe;
- secure;
- reliable;

- portable.

As we can read in "A Rationale-Based Classification of MISRA C Guidelines", Bagnara et al [4]:

> *"Each of the 175 guidelines of MISRA C is classified as being either a* ***directive*** *or a* ***rule****:*
> ***Rule****: a guideline such that information concerning compliance is fully contained in the source code and in the language implementation.*
> ***Directive****: a guideline such that information concerning compliance is not fully contained in the source code and language implementation: requirements, specifications, designs and other considerations may need to be taken into account.*
>
> *One of the things that is often misunderstood is that MISRA C is much more than just the set of its guidelines. The guidelines are meant to be used in the framework of a documented software development process.*
>
> *...*
>
> *The* ***deviation process*** *is an essential part of the adoption of the MISRA Guidelines, each one of which is assigned a single category:* ***mandatory****,* ***required*** *or* ***advisory****, defined as follows.*
>
> ***Mandatory****: C code that complies with MISRA C must comply with every mandatory guideline: deviation is not permitted.*
>
> ***Required****: C code that complies with MISRA C shall comply with every required guideline: a formal deviation is required where this is not the case.*
>
> ***Advisory****: these are recommendations that should be followed as far as it is reasonably practical: formal deviation is not required, but non-compliances should be documented.*
>
> *Whenever complying with a guideline goes against code quality or does not allow access to the hardware or does not allow integrating or use suitably qualified adopted code, the guideline has to be deviated. That is, instead of modifying the code to bring it into compliance, for required guidelines, a written argument has to be provided to justify the violation whereas, for advisory guidelines, this is not necessary."*

MISRA comprehends a number of rules that are focused mainly on these aspects:

- avoiding possible compiler differences;
- avoiding using functions and constructs that are prone to failure;
- produce maintainable and debuggable code;
- best practices in general;
- complexity limits.

At the moment of this writing there have been three releases of the MISRA C standard:

- MISRA C:1998;
- MISRA C:2004;
- MISRA C:2012.

A small selection from the MISRA C 2012 guidelines follows, in order to convey the overall idea of the coding standard:

Rule 15.6:

The body of an iteration-statement or a selection-statement shall be a compound-statement

Category: Required
Analysis: Decidable, Single Translation Unit
Applies to: C90, C99

It is possible for a developer to mistakenly believe that a sequence of statements forms the body of an iteration-statement or selection-statement by virtue of their indentation. The accidental inclusion of a semi-colon after the controlling expression is a particular danger, leading to a null control statement. Using a compound-statement clearly defines which statements actually form the body. Additionally, it is possible that indentation may lead a developer to associate an else statement with the wrong if.

A famous case in which such a rule would have prevented problems is the "Apple goto fail vulnerability": on 2014-02-21 Apple released a security update for its implementation of SSL/TLS in many versions of its operating system and the indicted code is reported in Listing 3.1.

> *"The problem was the second (duplicate) "goto fail". The indentation here is misleading; since there are no curly braces after the "if" statement, the second "goto fail" is always executed. In context, that meant that vital signature checking code was skipped, so both bad and good signatures would be accepted. The extraneous "goto" caused the function to return 0 ("no error") when the rest of the checking was skipped; as a result, invalid certificates were quietly accepted as valid." [14]*

```
1 if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
2   goto fail;
3   goto fail;
4 ... other checks ...
5 fail:
6   ... buffer frees (cleanups) ...
7   return err;
```

Listing 3.1: The Apple goto fail vulnerability

Rule 3.2:

Line-splicing shall not be used in // comments

Category: Required
Analysis: Decidable, Single Translation Unit
Applies to: C99

If a // commented line ends with a back-slash followed by a new-line, then the following line is part of the comment, and, if this was not intended, an important line of code maybe lost. The following example shows how a seemingly-innocuous path separator at the end of the comment may accidentally comment out the next line of code. [2]

```
1 // see critical.* in c:\project\src\
2 critical_function(); /* Non-compliant */
3
4 // see critical.* in c:\project\src
5 critical_function(); /* Compliant */
```

Listing 3.2: Examples of MISRA C 2012 Rule 3.2 violation and compliance

Rule 9.1:

The value of an object with automatic storage duration shall not be read before it has been set

Category: Mandatory
Analysis: Undecidable, System
Applies to: C90, C99

Note that array elements and structure members are considered as discrete objects and they must be (recursively) initialized. The rationale is that, according to the C Standard, objects with automatic storage duration are not automatically initialized and can therefore have indeterminate values. Reading them while their value is indeterminate is undefined behavior. [2]

Dir 4.4:

Sections of code should not be "commented out"

Category: Advisory
Applies to: C90, C99

This rule applies to both // and /* ... */ styles of comment. Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. #if or #ifdef constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

Chapter 4

Language Server Protocol

Implementing support for features like autocomplete, refactoring, navigating to a symbol's definition, syntax highlighting, error and warning markers, or documentation on hover for a programming language is a significant effort. Traditionally this work must be repeated for each development tool, as each provides different APIs for implementing the same features. The idea behind a Language Server is to provide the language-specific smarts inside a server that can communicate with the development tooling over a protocol that enables inter-process communication. The Language Server Protocol was originally developed for Microsoft Visual Studio Code and is now an open standard. On 2016 June 27, Microsoft announced a collaboration with Red Hat and Codenvy to standardize the protocol's specification and it is now hosted and developed on GitHub. The idea behind the Language Server Protocol (also called LSP from now on) is to standardize the protocol used by tools and servers to communicate, so a single Language Server can be reused in multiple development tools, which in turn can support new languages with minimal effort.

As can be observed in the diagram, a Language Server runs as a separate process and development tools communicate with the server using the LSP over JSON-RPC.

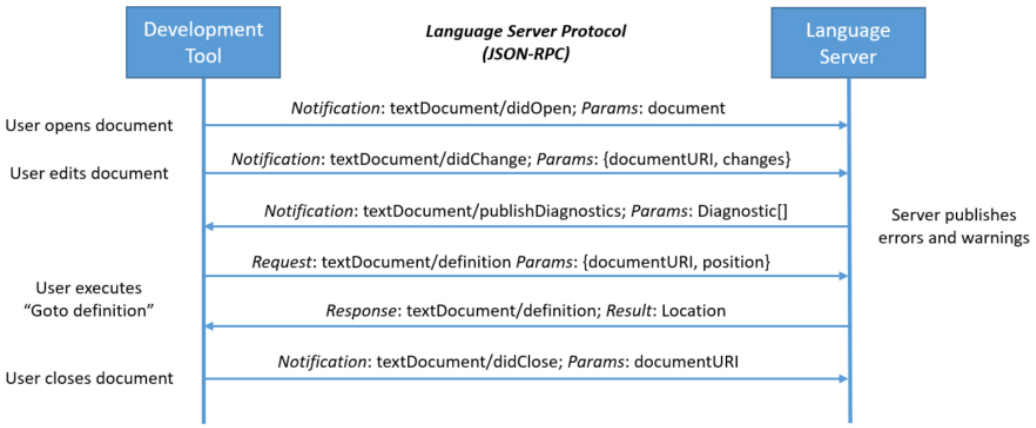


Figure 4.1: An example of communication between a tool and a Language Server through Language Server Protocol [11]

4.1 The protocol

The last version of the LSP at the time of this writing and the one that has been used in the proof-of-concept is 3.17. The base protocol consists of a header and a content part (similar to HTTP). The header and content part are separated by a '\r\n'. The header part consists of header fields which conform to the HTTP semantics. Currently the only header fields supported are Content-Length and Content-Type. The content part, instead, contains the actual content of the message. This is used to send messages over JSON-RPC to describe requests, responses and notifications.

```
Content-Length: ...\r\n
\r\n
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/didOpen",
  "params": {
    "textDocument": {
      "uri": "file:///home/test-user/project/main.c",
      "languageId": "c",
      "version": 1.
      "text": "int main() { ... }"
    }
  }
}
```

Listing 4.1: textDocument/didOpen notification example

```
Content-Length: ...\r\n
```

```
2 \r\n
3 {
4   "jsonrpc": "2.0",
5   "id": 1,
6   "method": "workspace/executeCommand",
7   "params": {
8     "command": "trigger-analysis",
9     "arguments": ["file:///home/test-user/project/main.c"]
10  }
11 }
```

Listing 4.2: workspace/executeCommand request example

Not every Language Server can support all features defined by the protocol. LSP therefore provides *capabilities*, so that both the development tool and the Language Server can announce their supported features using capabilities at the beginning of the communication (during the "initialize" request). Once the capabilities of each participant in the communication are exchanged, the client and the Language Server can begin to send over requests, responses and notifications accordingly.

4.2 Document Synchronization

Each LSP client supports "textDocument/didOpen", "textDocument/didChange" and "textDocument/didClose" notifications. These are used by the Language Server to keep an internal synced version of the files the user is viewing. This allows the Language Server to react to changes to the file and reference specific parts of the document. For example, whenever the notification "textDocument/didChange" is received we can safely assume that parts of the file referenced in the notification have been changed. The LSP specification refers to three kind of document synchronization:

- none, disabling all kind of document syncing;
- full, which means that documents are synced by always sending the full content;
- incremental, which means that, after sending the full content on file opening, only incremental updates to the document are sent.

In the scope of this thesis, the notification "textDocument/didSave" has been largely used. Below its params interface structure can be observed.

```
interface DidSaveTextDocumentParams {
  /**
    * The document that was saved.
    */
  textDocument: TextDocumentIdentifier;

  /**
    * Optional the content when saved. Depends on the
     includeText value
    * when the save notification was requested.
    */
  text?: string;
}
```

Listing 4.3: DidSaveTextDocumentParams interface [11]

4.3 Language features

Language Features provide the actual smarts in the Language Server Protocol. Usually executed on a [text document, position] tuple, the main language feature categories are:

- code comprehension features like hover or goto definition;
- coding features like diagnostics, code complete or code actions.

Diagnostics in particular have been the main feature used in the context of this project. In the first conceptualization of the LSP, they were something that the server could only publish to the client which were then "owned" by the server only, so that they were its responsibility to clear if necessary. When a file changes, it is the server's responsibility to re-compute diagnostics and push them to the client. This approach has the advantage that for workspace-wide diagnostics the server has the freedom to compute them at a server preferred point in time.

However, with such an approach, the server can't prioritize the computation for the file in which the user types or which are visible in the editor. Therefore, the concept of diagnostic pull requests was introduced in order to give a client more control over the documents for which diagnostics should be computed and at which point in time.

```
export interface Diagnostic {
  /**
   * The range at which the message applies.
   */
  range: Range;
```

```
6
7   /**
8    * The diagnostic's severity. Can be omitted. If omitted it is
       up to the
9    * client to interpret diagnostics as error, warning, info or
       hint.
10   */
11  severity?: DiagnosticSeverity;
12
13  /**
14   * The diagnostic's code, which might appear in the user
       interface.
15   */
16  code?: integer | string;
17
18  /**
19   * An optional property to describe the error code.
20   *
21   * @since 3.16.0
22   */
23  codeDescription?: CodeDescription;
24
25  /**
26   * A human-readable string describing the source of this
27   * diagnostic, e.g. 'typescript' or 'super lint'.
28   */
29  source?: string;
30
31  /**
32   * The diagnostic's message.
33   */
34  message: string;
35
36  /**
37   * Additional metadata about the diagnostic.
38   *
39   * @since 3.15.0
40   */
41  tags?: DiagnosticTag[];
42
43  /**
44   * An array of related diagnostic information, e.g. when
       symbol-names within
45   * a scope collide all definitions can be marked via this
       property.
46   */
47  relatedInformation?: DiagnosticRelatedInformation[];
48
49  /**
```

```
50     * A data entry field that is preserved between a
51     * 'textDocument/publishDiagnostics' notification and
52     * 'textDocument/codeAction' request.
53     *
54     * @since 3.16.0
55     */
56    data?: unknown;
57 }
```

Listing 4.4: Diagnostic interface [11]

Part II

The Proof of Concept

In this part of the thesis, the main results of our work are presented.

First, in Chapter 5, the starting point is presented: current tools and ECLAIR reports fruition are analyzed.

Chapter 6 briefly describes the first experiment with the Language Server Protocol and ECLAIR, detailing the issues we faced. It laid the foundations for the prototype described in Chapter 7.

In Chapter 7, the architecture of the project is presented. After a first high-level view of the interactions between the components, each of them is presented and analyzed in detail.

Chapter 5

Starting point

Before moving to an in-depth analysis of what we realized, it is important to understand the starting point from which the idea of such a project was born. ECLAIR is already widely used in many contexts in which static analysis is mandatory and enforced by standard regulations. Currently, the developers run ECLAIR on their desktop through a GUI or a CLI.

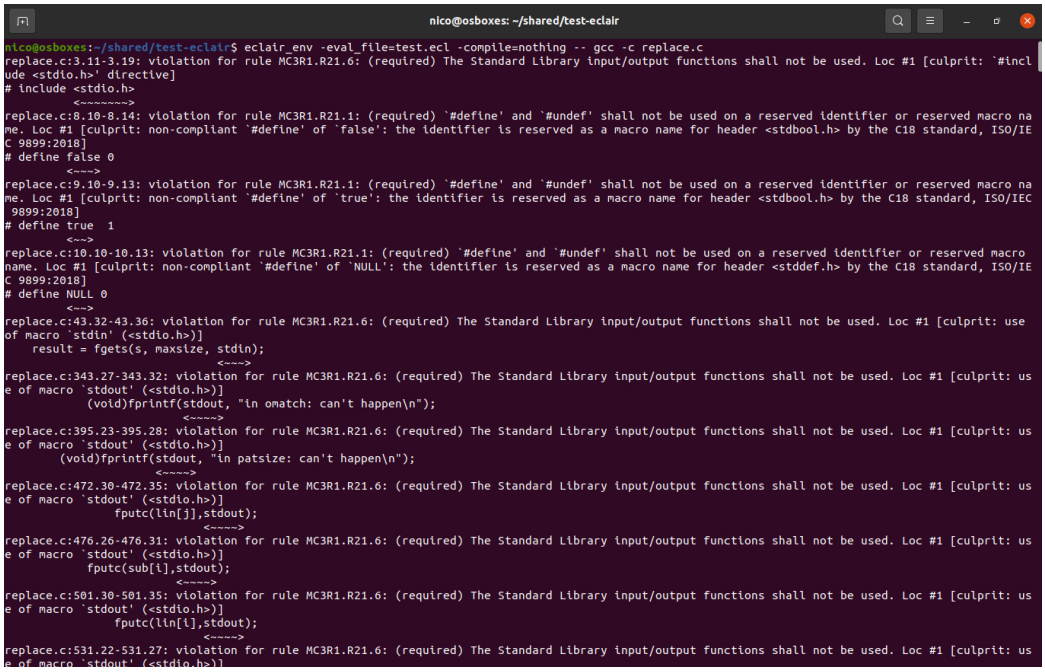


Figure 5.1: ECLAIR CLI

In particular the first step in ECLAIR setup is configuring the toolchain, that means to indicate installed tools such as compilers and linkers: this is

required since the kind of analysis that is performed is bound to the way the software is compiled.

In addition, ECLAIR needs a *.ecl* file in which rules can be activated or disabled and the report format is specified.

```
1 -project_root=getenv("ECLAIR_PROJECT_ROOT")
2
3 -enable=B.REPORT.TXT
4 -enable=MC3R1.R
5 -disable='sav&&!B'
6 # -enable=MC3R1.R8.13
7
8 -frames={hide,'kind(object||program||project)'}
9
10 # Hides all reports that have all areas external to project root
      tree.
11 -reports+={hide,all_exp_external}
```

Listing 5.1: Example *.ecl* file

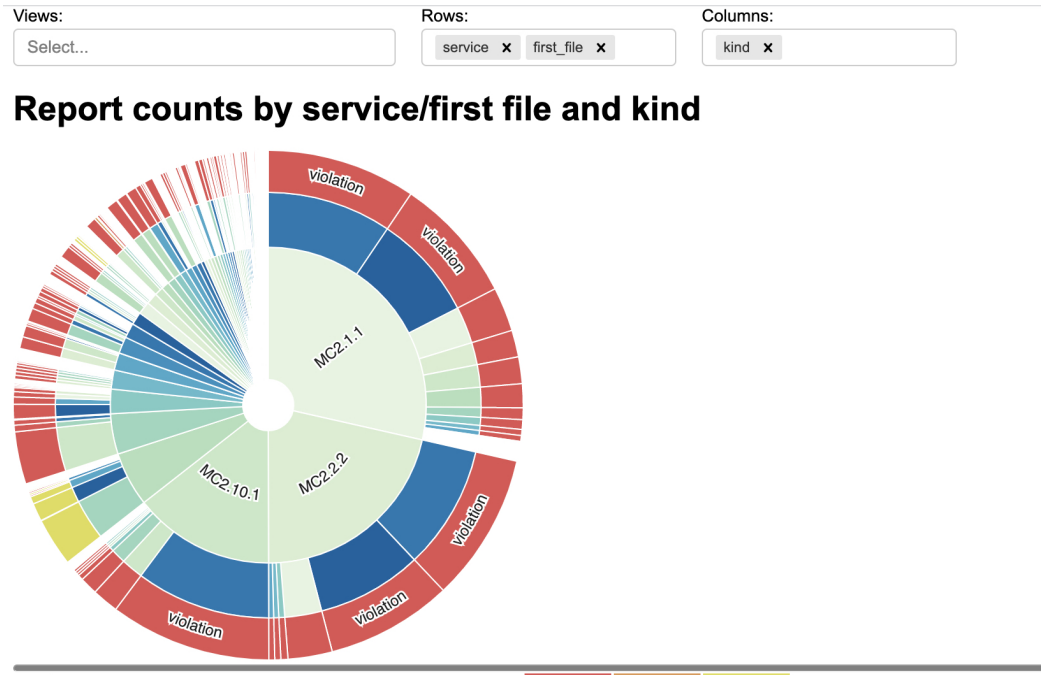
After the analysis, ECLAIR can produce different kinds of outputs:

- summary outputs, which contain comprehensive information about the analysis, as well as counts of the issues uncovered by the analysis per service, per file, per tag and combinations;
- rich outputs, which contain comprehensive information about the analysis results without going into the detail of each reported program condition;
- detailed outputs, which contain comprehensive information about the analysis results, including all details about each reported program condition (such as a coding rule violation or a possible run-time error) and all the information required for a proper understanding of each individual issue reported by the analysis;
- metric outputs, which contain the values of the metrics collected for each file, function and project.

These outputs can be returned in different formats:

- summary and rich outputs in printable format (*.odt* or *.doc*), HTML format or pure text;
- rich interactive outputs in spreadsheet format, which contain the set of ECLAIR findings in a form that is suitable for the communication to third parties that are only interested in the residual violations and whether they have been justified and how;

- detailed outputs in interactive spreadsheet format, pure text format or printable format.



service first_file	violation	caution	information
TOTAL	**7302**	**45**	**491**
B.EXPLAIN			**1**
_SPECIAL/_project qpc_4.5.02a_MC2_matrix_			1
MC2.1.1	**2236**		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/bsp.c	48		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/core_c…	58		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/core_c…	133		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/core_c…	133		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/lm3s_c…	627		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/startup…	119		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/cmsis/system…	29		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/display96x16…	741		
examples/arm-cortex/qk/gnu/dpp-qk-ev-lm3s811/display96x16…	30		

Figure 5.2: Example of ECLAIR rich output in HTML format

Since static analysis integration in continuous integration pipelines is a must-have in modern development, the eclairit.com web site was created. This demonstrates the use of ECLAIR integrated into Jenkins[1] and GitLab. An integration server performs the analysis and provides the reports directly in the browser, without having to install ECLAIR on each PC. This was the first change of paradigm from the traditional analysis performed directly on

[1]https://www.jenkins.io

developers' machines. This tool gave us the inspiration to integrate ECLAIR in the development workflow in a new way: we wanted to perform the analysis while the users were writing the code.

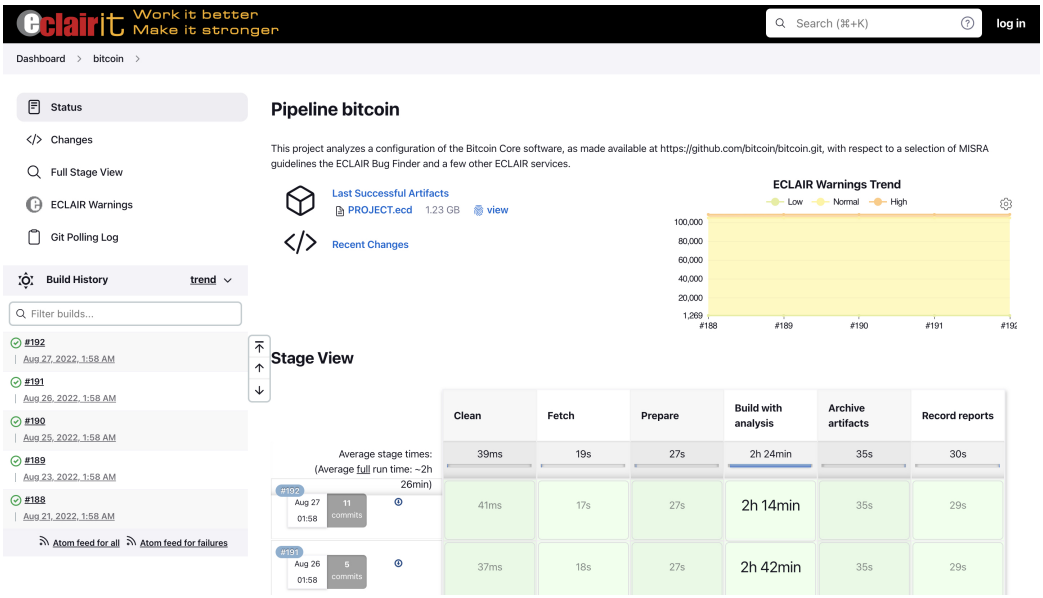


Figure 5.3: eclairit.com

Chapter 6

The first experiment

When we first thought about using the LSP in this project we decided to try it first hand with a simple implementation of the ECLAIR analysis in the IDE. The first iteration consisted of a Language Server which simply listened to file changes which would in turn trigger a new ECLAIR analysis each time using the CLI. Then the analysis output would be parsed, converted to LSP Diagnostics and sent over to the client.

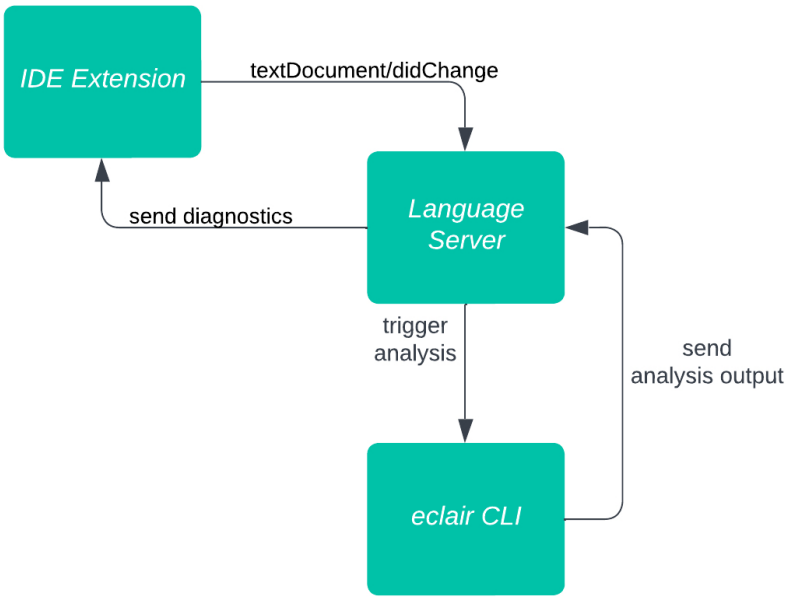


Figure 6.1: First experiment with the Language Server Protocol

In order to fully understand the benefits the LSP could give us, we realized two extensions, for VSCode and Sublime Text. In both cases we could observe that, with just a few lines, we were able to see the violations marked in the file.

```
{
  "clients": {
    "elcair": {
      "command": [
        "eclair-server",
        "--stdio"
      ],
      "enabled": true,
      "languages": [
        {
          "languageId": "c"
        }
      ]
    }
  },
  "log_debug": true
}
```

Listing 6.1: First Sublime Text working configuration

```
const { LanguageClient } = require("vscode-languageclient")

module.exports = {
  activate(context) {
    const executable = {
      command: "eclair-server",
      args: ["--stdio"]
    }

    const serverOptions = {
      run: executable,
      debug: executable
    }

    const clientOptions = {
      documentSelector: [{
        scheme: "file",
        language: "c"
      }]
    }

    const client = new LanguageClient(
      "eclair-extension-id",
      "Eclair",
```

```
25          serverOptions,
26          clientOptions
27      )
28
29      context.subscriptions.push(client.start())
30    }
31 }
```

Listing 6.2: First VSCode working extension

Chapter 7

Project architecture

This chapter presents the prototype we realized and the reasons behind some choices. The main components analyzed here are:

- *eclair*;
- *eclair_report*;
- a Language Server;
- the VSCode extension.

The following diagram aims at giving a high-level overview of the components and their interactions: each process will be analyzed in-depth in the following sections.

However, it is already possible to understand the neat division between the IDE extension, the Language Server and the analysis services. Through the LSP, the IDE communicates to the Language Server, which in turn invokes the *eclair* CLI to perform the analysis or asks *eclair_report* for interesting violations given the current viewport.

The Language Server uses the output received from *eclair_report* to communicate to various IDE extensions, through the LSP, what must be shown to the end user.

The separation of concerns and the decoupling of the components is at the foundation of this architecture: *eclair_report* is expected to care only about serving the violations to the Language Server, while the *eclair* CLI performs the analysis in an agnostic way regarding how the violations will be served. The only interface the IDE extension has to deal with is the Language Server, which is always the same, independently from the IDE, and hides the complexity of implementing the calls to the analysis services. This way, building new extensions for other IDEs is painless and, with just a

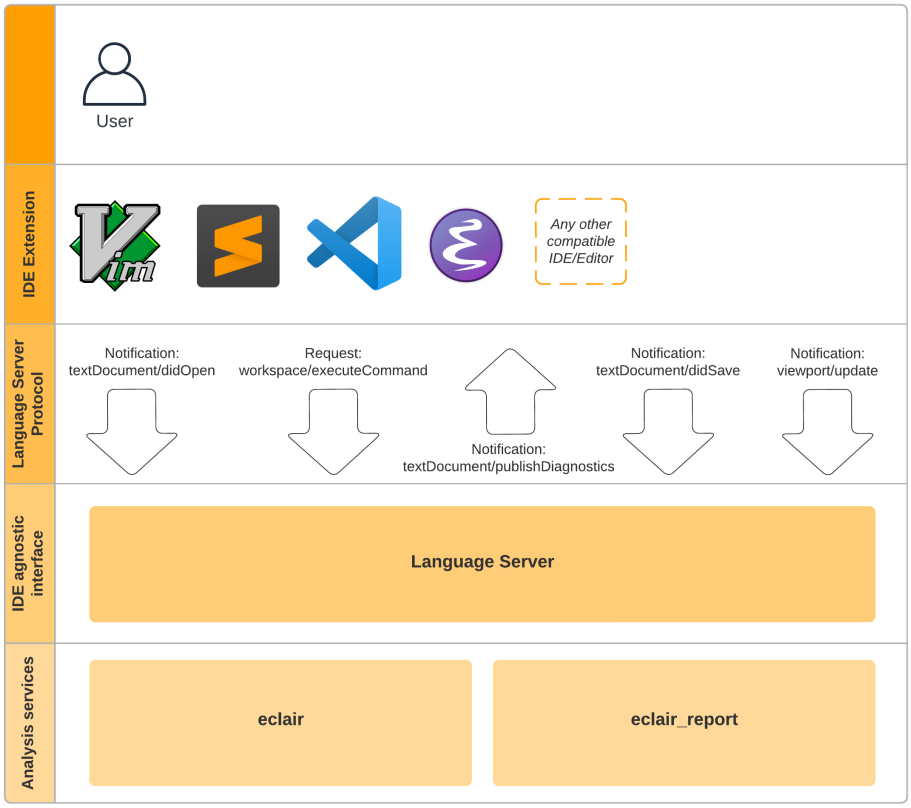


Figure 7.1: Project architecture

few lines of code, it is possible to communicate with the Language Server from the earlier stages of the development. On top of these foundations, extension developers can concentrate on providing the best experience to the end users without worrying about linking the analysis tools to the dishomogeneous IDEs' primitives and APIs.

7.1 *eclair*

The *eclair* CLI is used to invoke the static analyzer. In order for ECLAIR to work, there must exist an *.ecl* file that specifies:

- project-specific configurations;
- MISRA rules that should be checked for compliance;

- report output format.

So the first thing before proceeding with the analysis, is to check for an existing *.ecl* file in the project directory.

There are three occasions in which the Language Server invokes the *eclair* CLI:

- a new file is opened for the first time;
- changes to a file are saved (the user can opt out of this feature);
- an analysis is manually triggered.

The first two cases are handled automatically whenever the Language Server receives the corresponding notifications, respectively *"textDocument/didOpen"* and *"textDocument/didSave"*. The third case, instead, relies on a *"workspace/executeCommand"* request that is sent from the IDE extension to the Language Server.

After performing an analysis, the *eclair* CLI sends its findings to *eclair_report* which in turn saves them into a project internal database, in *.ecd* format. This database will then be used by *eclair_report* to provide the violations.

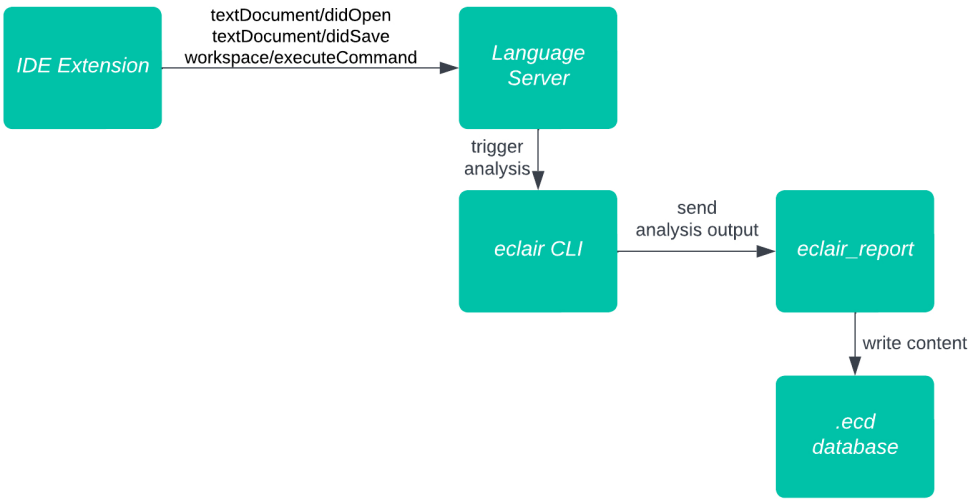


Figure 7.2: Interaction with the *eclair* CLI

7.2 *eclair_report*

eclair_report is already part of the ECLAIR ecosystem: its job is to export ECLAIR analysis findings to different formats (more details about the currently available formats can be found in Chapter 5). In the first experiment, described in Chapter 6, the analysis was performed from the beginning whenever the user changed something in the file and violations were returned all at once: unfortunately we noticed that this approach resulted in a slow feedback to the user and sometimes useless analyses were performed.

We decided to use a combination of *eclair* and *eclair_report* to perform the analysis only when necessary and serve its output incrementally. The first thing *eclair_report* has to do in our architecture is to save the analysis findings in the project level *.ecd* database.

Once we had established this, we could build on top of this database features like incrementality and various optimization for the following analyses. After some reasoning, we decided to proceed with the first, and postpone for now the latter.

Returning all the violations all at once to the IDE wasn't optimal in particular for huge files: we had to find a new way to navigate them. We designed a new feature of *eclair_report* that would allow us to interrogate the service and ask incrementally the violations recorded: when interrogated, the tool would be able to receive a top line index and bottom line index and return only the violations in the given range. Hence, keeping an updated internal state in the Language Server with the current user viewport, we could achieve the incrementality we aimed at.

Figure 7.3: Interaction with *eclair_report*

7.3 Language Server

The Language Server is a software written in Typescript, a superset of JavaScript that adds optional static typing to the language, that runs on NodeJS and behaves like a bridge between the ECLAIR ecosystem and the IDE.

In particular, it relies on an implementation of the LSP provided as an NPM library from Microsoft, vscode–languageserver[1], and makes the calls to *eclair* and *eclair_report*.

The Language Server saves an internal representation of the document the user is looking at, incrementally updates it during the editing, and uses it to mark violations. Every time the user saves the file, opens a new one or manually triggers an analysis a message is sent over to the Language Server, which in turn will trigger an execution by the *eclair* CLI. To avoid useless invocations, the file savings are batched for a defined period of time before making the call asking to perform the analysis.

```
type Viewport = {
  filename: string
  topLineIndex: number
  bottomLineIndex: number
}

let viewport: Viewport
const BUFFER = 20

connection.onNotification((method, params) => {
  if (method === "viewport/update") {
    viewport = params as Viewport

    const uri = "file://${viewport.filename}"

    connection.sendDiagnostics({
      uri,
      diagnostics: eclairReports.getViolations(workspaceFolder,
    uri, viewport.topLineIndex - BUFFER, viewport.bottomLineIndex
    + BUFFER)
    })
  }
})
```

Listing 7.1: Server side code for viewport synchronization

```
let lastRecorded = {
  filename: null,
```

[1] https://www.npmjs.com/package/vscode-languageserver

```
3   topLineIndex: null,
4   bottomLineIndex: null
5 }
6
7 let timeout: NodeJS.Timeout
8 Window.onDidChangeTextEditorVisibleRanges(() => {
9   if (timeout) {
10    clearTimeout(timeout)
11  }
12  timeout = setTimeout(() => {
13    const textEditor = Window.activeTextEditor
14    if (textEditor) {
15      const visibleRange = textEditor.visibleRanges[0]
16      const filename = textEditor.document.fileName
17      const topLineIndex = visibleRange.start.line
18      const bottomLineIndex = visibleRange.end.line
19      if (
20        lastRecorded.filename !== filename ||
21        lastRecorded.topLineIndex !== topLineIndex ||
22        lastRecorded.bottomLineIndex !== bottomLineIndex
23      ) {
24        lastRecorded.filename = filename
25        lastRecorded.topLineIndex = topLineIndex
26        lastRecorded.bottomLineIndex = bottomLineIndex
27        client.sendNotification("viewport/update", lastRecorded)
28      }
29    }
30  }, 500)
31 })
```

Listing 7.2: VSCode extension side code for viewport synchronization

One of the sophistications we had to adopt is the incrementality in the diagnostics: we could not send the violations all at once. However, in order to visualize only the necessary ones, the Language Server must know the file the user is currently looking at and the lines currently in the viewport.

Unfortunately this feature is not natively offered by the LSP. Thus, inspired from the Text Document Synchronization, we opted for the following approach: whenever the user changes the current active file or viewport, the client retrieves these informations using the IDE's API and uses an LSP notification to send these details to the server, which in turn keeps an internal state. Hence, the Language Server is able to interrogate *eclair_report* and receive only the violations for the file and the lines the user actually needs.

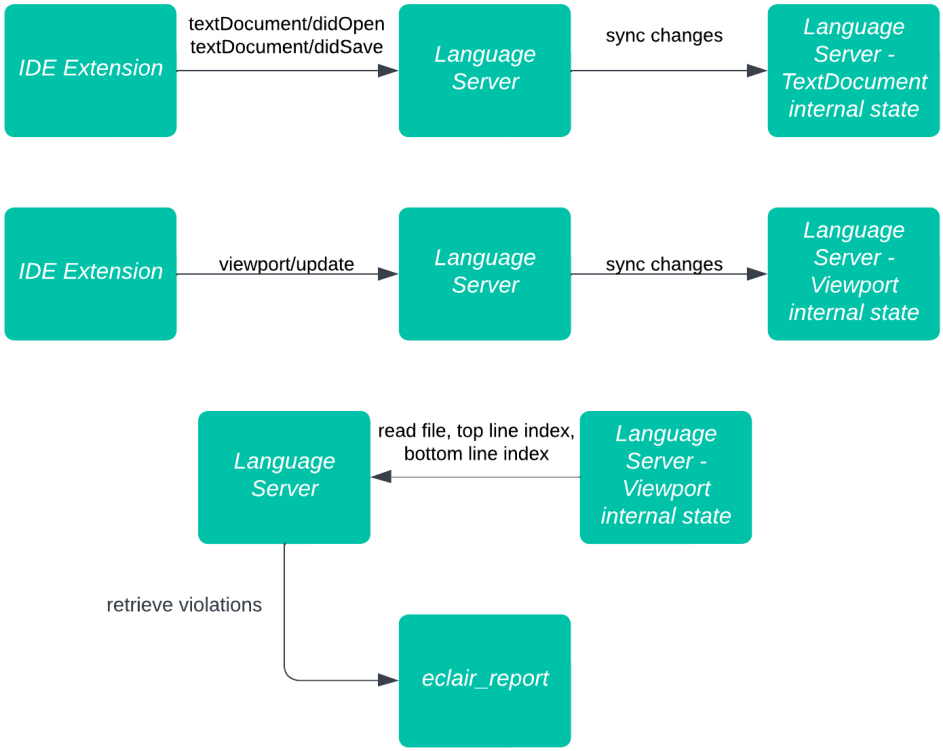


Figure 7.4: Interactions with the Language Server

7.4 VSCode extension

The VSCode extension, which in this case exemplifies all the possible IDE plugins/extensions that can be realized on top of this system, has nearly no logic implemented. Its main task is the creation of the connection with the Language Server.
In addition to that, some other features, not completely agnostic to the IDE, have been implemented: aside from the viewport synchronization logic, which must retrieve the current viewport using IDE's primitives, already discussed in Section 7.3, we also had to implement a way to manually trigger the analysis and give the user the possibility to opt out from the "analysis on save" feature. Regarding the manually triggered analysis, the Listing 7.3 should give more insights about how we linked the VSCode "Commands.registerCommand" API to the "workspace/executeCommand" request.

```
1 Commands.registerCommand("eclair.triggerAnalysis", () => {
2   const uri = Window.activeTextEditor.document.uri
3   client.sendRequest("workspace/executeCommand", {
```

```
4       command: "trigger-analysis",
5       arguments: [uri.toString()]
6     } as ExecuteCommandParams)
7 })
```

Listing 7.3: VSCode extension triggerAnalysis command

On the other hand, the automatic "analysis on save" was something we had to think through: a lot of IDEs users have autosaving configurations enabled, so we had to at least make possible to opt out of this feature, given how expensive the analysis is. We exposed a boolean configuration, *eclair.performAnalysisOnSave*, to switch off the "analysis on save" feature and, to fully comply with the LSP approach, we decided that the IDE extension should send over this information with the capabilities in the initialization phase. In particular, as can be seen on line 14 of Listing 7.4, all the content of *Workspace.getConfiguration("eclair")* is sent over: if the server is able to use the forwarded settings it will, otherwise they will simply be ignored.

```
1 const debugOptions = { execArgv: ["--nolazy", "--inspect=6011"]
    }
2 const serverOptions = {
3   run: { module, transport: TransportKind.ipc },
4   debug: { module, transport: TransportKind.ipc, options:
    debugOptions }
5 }
6 const clientOptions: LanguageClientOptions = {
7   documentSelector: [
8     { scheme: "file", language: "c", pattern: "${folder.uri.
    fsPath}/**/*" }
9   ],
10  diagnosticCollectionName: "eclair-language-server",
11  workspaceFolder: folder,
12  outputChannel: outputChannel,
13  progressOnInitialization: true,
14  initializationOptions: Workspace.getConfiguration("eclair")
15 }
16 const client = new LanguageClient("eclair-vscode-client", "
    ECLAIR", serverOptions, clientOptions)
17 client.start()
```

Listing 7.4: VSCode extension initialization

Extensibility is at the core of the VSCode plugin and, thanks to the LSP notification based communication, it's simple and intuitive to implement new features, even at different time, on the server and on the client. In around 200 lines of code, our extension was fully functional and ready to be used as an inspiration to develop other extensions that use the same Language

Server.

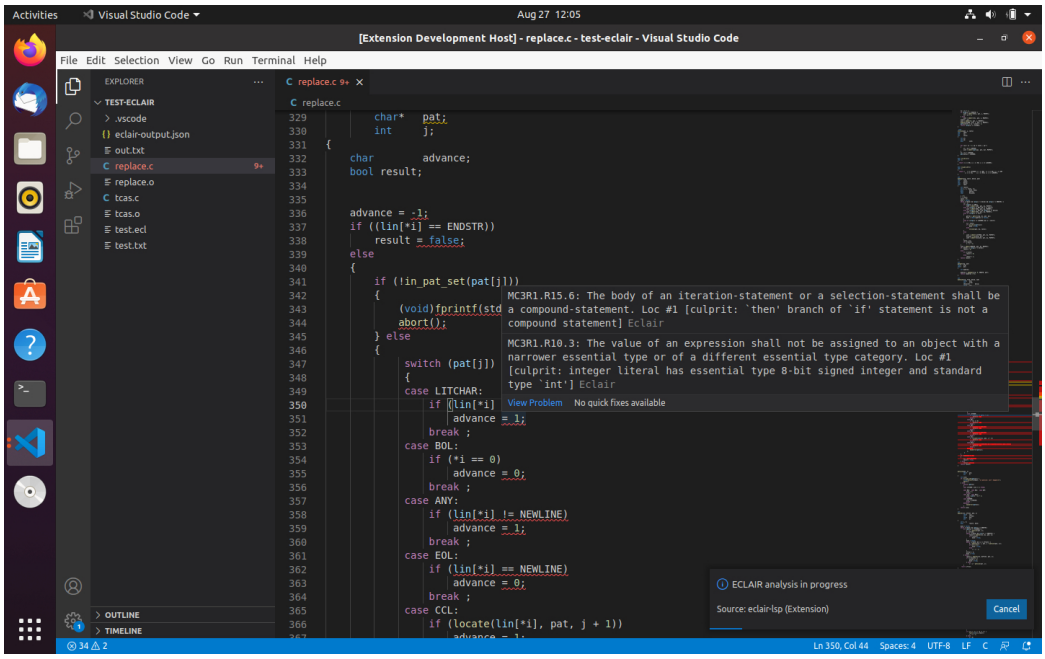


Figure 7.5: A screenshot of the VSCode extension in action

Part III

Conclusion

Finally, in Chapter 8 some conclusions are drawn. After an overview of the challenges we faced, some reasonings about future work are presented.

Chapter 8

Discussion and future work

The work we did to implement this proof-of-concept gave us a taste of the benefit a tool of this kind can be to developers: after the first analysis was performed, we could navigate smoothly through the code and receive almost instant feedback about the violations that were detected. The aim of this project was to determine whether this could be a feasible way to use ECLAIR analysis output and to lay the foundations of what can be a game changing tool for developers who develop safety-critical software.

During the realization, we faced issues common to this kind of projects and found appropriate solutions: first the need for a common interface between the editors was solved using the LSP, then we relied on *eclair_report* to answer the need for incrementality in the showing of the violations, then we achieved parallelism spawning multiple ECLAIR analysis from the Language Server when necessary and so on.

We also had time to think about future improvements, that can make this piece of software production-ready and experiments to improve it: for example the pull diagnostics mechanism should be tested, and see if it actually could replace the current one, which pushes diagnostics from the server to the client.

This approach would allow the client to ask for violations only when it decides they are needed and not receive them passively from the server.

Another improvements on the side of the ECLAIR analyzer would be to reuse the previous analyses and re-analyze only what actually changed, without having to analyze the entire file. While this is feasible for some kinds of violations, it can be challenging for others.

At last, thanks to the decoupling granted by the Language Server Protocol, we can imagine scenarios in which the analysis is not even performed

on the developer machine, but instead on a dedicated one that sends the results of the analysis through the protocol. At the moment of this writing the transport channel can either be stdio, sockets, named pipes, or node ipc if both the client and server are written for Node.js.

The leitmotif of this thesis has been the simplicity: from the very beginning we had a clear view of the components, how we wanted them to interact and the precise level of isolation each level should have from the others that resulted in a simple and elegant implementation.

One of the reasons behind this project was to assess whether the Language Server Protocol, an ever-growing standard, could actually make it possible to integrate an existing analysis tool into IDEs easily. We think that, after developing the reusable Language Server, the only 200 lines of code that were necessary to integrate it into VSCode have been our witnesses.

This thesis describes the underlying ideas, the challenges and the solutions explored during the implementation of this prototype. We hope that the reader had the opportunity to learn new lessons, understand the potentiality of the new technologies we used and to better understand the evolutions of the static analysis ecosystem and its tooling.

Bibliography

[1] R. Bagnara, A. Bagnara, F. Biselli, M. Chiari, and R. Gori. Correct approximation of IEEE 754 floating-point arithmetic for program verification. *Constraints*, February 2022.

[2] R. Bagnara, A. Bagnara, and P. M. Hill. The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software. In A. Podelski, editor, *Static Analysis: Proceedings of the 25th International Symposium (SAS 2018)*, volume 11002 of *Lecture Notes in Computer Science*, pages 5–23, Freiburg, Germany, 2018. Springer International Publishing.

[3] R. Bagnara, A. Bagnara, and P. M. Hill. The MISRA C coding standard: A key enabler for the development of safety- and security-critical embedded software. In DESIGN&ELEKTRONIK, editor, *embedded world Conference 2019 — Proceedings*, pages 543–553, Nuremberg, Germany, 2019. WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany.

[4] R. Bagnara, A. Bagnara, and P. M. Hill. A rationale-based classification of MISRA C guidelines. In DESIGN&ELEKTRONIK, editor, *embedded world Conference 2022 — Proceedings*, pages 440–451, Nuremberg, Germany, 2022. WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany.

[5] R. Bagnara, M. Barr, and P. M. Hill. BARR-C:2018 and MISRA C:2012 (with Amendment 2): Synergy between the two most widely used C coding standards. In DESIGN&ELEKTRONIK, editor, *embedded world Conference 2021 DIGITAL — Proceedings*, pages 378–391, Nuremberg, Germany, 2021. WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany.

[6] R. Bagnara, M. Chiari, R. Gori, and A. Bagnara. A practical approach to verification of floating-point C/C++ programs with `math.h`/`cmath`

functions. *ACM Transactions on Software Engineering and Methodology*, 30(1), 2020.

[7] Roberto Bagnara, Michele Chiari, Roberta Gori, and Abramo Bagnara. A practical approach to verification of floating-point c/c++ programs with math.h/cmath functions. *ACM Trans. Softw. Eng. Methodol.*, 30(1), dec 2021.

[8] K. Beck, C. Andres, and E. Gamma. *Extreme Programming Explained: Embrace Change*. XP series. Addison-Wesley, 2004.

[9] Patrick Cousot. Abstract interpretation in a nutshell. `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`, 2008.

[10] Massimo Franceschet and Angelo Montanari. An introduction to model checking (slightly revised by angelo montanari). `https://users.dimi.uniud.it/~angelo.montanari/MCclasses.pdf`, 2019.

[11] Microsoft. Language server protocol, 2021.

[12] Larry Smith. Shift-left testing. *Dr. Dobb's J.*, 26(9):56–ff, sep 2001.

[13] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. Jit feedback - what experienced developers like about static analysis. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 64–6409, 2018.

[14] David A. Wheeler. The apple goto fail vulnerability: lessons learned, 2021.

Ringraziamenti

Vorrei dedicare questo spazio alle persone che mi hanno supportato nella redazione di questo lavoro di tesi.

Ringrazio il mio relatore Roberto Bagnara che mi ha seguito, con la sua infinita disponibilità, pronto a fornirmi strumenti, migliorie e suggerimenti utili ai fini della stesura dell'elaborato.

Non posso non menzionare i miei genitori e mio fratello che da sempre mi sostengono nella realizzazione dei miei progetti.

Un ringraziamento particolare va a Giovanni Bruno, Valerio Versace e Daniele Donelli, che prima come mentori e poi come soci in Soluzioni Futura, sono stati in grado di trasmettermi la loro passione per l'informatica e con i quali affronto sfide tecnologiche sempre nuove.

Ringrazio Claudia Chiarenza, per essere stata sempre presente, per il suo supporto e incoraggiamento ad ogni passo di questo percorso.

Ringrazio inoltre Abramo Bagnara e Simone Ballarin di BUGSENG per i momenti di confronto, i contributi nel design del progetto e per la realizzazione dell'ecosistema ECLAIR.

Grazie a tutti i miei colleghi di corso, per avermi sempre incoraggiato fin dall'inizio del percorso universitario e per tutti i momenti di spensieratezza.

Infine, vorrei dedicare questo traguardo anche a me stesso, che possa essere l'inizio di una lunga e brillante carriera professionale.