



Università
Ca' Foscari
Venezia

Master's Degree programme in Computer Science

Final Thesis

Ca' Foscari
Dorsoduro 3246
30123 Venezia

LiSA and ROS Static Analysis for Robotics

Supervisor

Prof. Pietro Ferrara

Graduand

Giacomo Zanatta

Matriculation number 859156

Academic Year

2022/2023

To all the universe

Abstract

This thesis will show how a static analysis tool can extract meanings from ROS source code. After a brief introduction to the theories that underlie static analysis, we will talk about LiSA, a static analysis framework maintained and developed by SSV (Software and System Verification) Research Group - a team of professors and researchers headquartered at the Ca' Foscari University of Venice - and more specifically we will learn what a front end for LiSA is, how we can write our one to analyze a specific program, and how LiSA works internally to produce meaningful results.

Extracting meaning from code by scratch is a challenging task: it requires a solid understanding of mathematical and computational theories like abstract interpretation and, of course, a deep knowledge of the language under analysis. The peculiarity of using LiSA is that these theories are abstracted away, providing an easy-to-use library.

We will show how we can use PyLiSA - a Python front end for LiSA - to develop domain-specific analysis straightforwardly. Our focus will be on the rclpy library, which is the Python client library for the ROS (Robot Operating System) ecosystem. We will also discover how to extend LiSA to perform static analysis of Python code using the rclpy library.

Contents

| | |
|--|-----------|
| 1 Introduction | 1 |
| 1.1 Bugs and Safety-Critical Systems | 4 |
| 1.2 Static Analysis for Robotics | 6 |
| 1.3 Thesis Structure | 8 |
| 2 Preliminaries | 11 |
| 2.1 Computers and Machine Language | 11 |
| 2.2 Programming Languages | 13 |
| 2.3 Compilers | 13 |
| 2.3.1 Lexer | 14 |
| 2.3.2 Parser | 14 |
| 2.4 Semantics of Program | 14 |
| 2.5 Program Analysis | 16 |
| 2.5.1 About Completeness and Soundness | 16 |
| 2.5.2 Program Analysis Techniques | 19 |
| 2.6 Static Analysis | 22 |
| 2.6.1 Abstract Interpretation | 24 |
| 2.6.2 Lattice and Partial Orders | 25 |
| 3 LiSA | 29 |
| 3.1 Introduction to LiSA | 29 |
| 3.2 Project Structure | 30 |
| 3.2.1 LiSA Submodules | 30 |
| 3.3 PyLiSA, Front Ends, and Antlr | 34 |
| 3.3.1 Antlr | 34 |
| 3.3.2 Front Ends and PyLiSA | 34 |
| 3.4 LiSA's Control Flow Graph | 39 |
| 3.4.1 About Statements | 39 |
| 3.4.2 About Edges | 42 |

| | |
|--|-----------|
| 3.4.3 Typing | 43 |
| 3.4.4 PyLiSA CFG | 44 |
| 3.5 LiSA's Analysis | 44 |
| 3.5.1 The Fundamentals | 44 |
| 3.5.2 Abstract State | 47 |
| 3.5.3 Analysis State | 48 |
| 3.5.4 The CallGraph | 49 |
| 3.5.5 Semantics of Statements | 51 |
| 3.6 Architectural Scheme | 51 |
| 3.7 Checkers | 52 |
| 3.7.1 How they Works | 53 |
| 3.7.2 An Example | 54 |
| 3.8 PyLiSA SABL | 56 |
| 3.9 Running a LiSA Analysis | 58 |
| 4 ROS | 61 |
| 4.1 Introduction to ROS | 62 |
| 4.1.1 Brief Definition of DDS | 63 |
| 4.2 Concepts and Terminology | 64 |
| 4.2.1 The ROS Domain | 64 |
| 4.2.2 The ROS Graph | 64 |
| 4.2.3 Nodes | 64 |
| 4.2.4 Topics | 64 |
| 4.2.5 Messages | 65 |
| 4.2.6 Parameters | 65 |
| 4.2.7 Services | 65 |
| 4.2.8 Actions | 65 |
| 4.2.9 Discovery Process | 66 |
| 4.3 ROS API Architecture | 67 |
| 4.4 About DDS-Security and SROS2 | 68 |
| 4.5 rclpy | 71 |
| 4.5.1 Application Life Cycle | 71 |
| 4.5.2 The rclpy Module | 72 |
| 4.6 How to Use rclpy: an Example | 75 |
| 5 Static Analysis for ROS (An Introduction) | 81 |
| 5.1 State of the Art of Static Analysis for Robotics | 81 |
| 5.2 Why this Thesis Exists | 82 |

| | |
|---|------------|
| 6 Lisa and ROS: Static Analysis for Robotics | 85 |
| 6.1 Introduction | 85 |
| 6.2 PyLiSA Front End Extensions | 87 |
| 6.2.1 Object Declaration | 87 |
| 6.2.2 Object Inheritance Support | 88 |
| 6.2.3 String Constant Propagation | 90 |
| 6.3 Analysis of the rclpy Library | 90 |
| 6.3.1 rclpy Semantics | 90 |
| 6.3.2 LiSA Analysis Configuration | 95 |
| 6.3.3 Preliminary Results of the Analysis | 97 |
| 7 Future Works | 107 |
| 8 Conclusion | 111 |
| Acknowledgements | 113 |
| Bibliography | 115 |

Chapter 1

Introduction

Imagine that you are working as a developer for a big consultant IT company. It's 9.00 AM on a cold Monday morning, and you haven't already recovered from the hangover of your crazy Saturday night. Halfheartedly, you walk to your desk with your coffee mug, thanking the creator of Smart Working. You open your Teams app and find out that the Project Manager has already started the Daily Meeting.

Quaffing your coffee, you enter the call, and, long story short, Project Manager told the team that she received an angry email from the client stating that the customer account page of his e-commerce site prints out a hideous internal error page.

After hearing these words, a slight inkling of panic starts to course through your backbone: in the last sprint, you changed the way system retrieves customer orders from the e-commerce cloud API and the manner in which these orders are being propagated to the front end.

The Project Manager told you to fix the problem ASAP and that a hotfix is required in order not to impact the current sprint tasks and stories and to make the system usable again.

Right after the end of the meeting, you immediately open your IDE to find out what piece of fantastic code you wrote, and you come upon this javascript code snippet:

```
1 function orderedProducts(apiOrders) {
2     var products = [];
3     for (var i = 0; i < apiOrders.length; i++) {
4         var order = apiOrders[i];
5         for (var i = 0; i < order.products.length; i++) {
6             products.push(order.products[i]);
```

```
7         }  
8     }  
9     return products;  
10 }
```

Code 1.1: A bugged javascript function

The function in code [1.1](#) takes in input an array of the order API and should return all the ordered products in all the orders. This function will not work as intended since you are redeclaring a global scope variable¹, *i*, (line 5). This could lead to an unpredicted behavior, such as an endless loop execution: suppose a customer has only placed three orders, with one product each.

At the first iteration of the outer loop, $i=0$. Redeclaring the *i* global variable in the inner loop declaration leads to overwriting the previous assignment on this variable. After the execution of the nested loop, $i=1$. The *i* variable is then incremented by 2, and the order in position 2 of the `apiOrders` array is accessed. The inner loop, again, will redeclare the *i* variable, setting it to 0, and after the execution, since this order has only one product, *i* will be equal to 2, and we process order 2 again until the javascript virtual machine crash.

In an optimal scenario, following sprint best practices [\[37\]](#) and performing SIT (System Integration Testing) or UAT (User Acceptance Testing) correctly, this type of bug will be intercepted during the sprint life cycle and appropriately handled in time. But sometimes, clients' requirements are not well-defined, and some tasks could be insidious to be tested. Furthermore, sometimes for one reason or another, tasks are underestimated. This leads developers to be in a hurry to complete the task in time, bringing stress and potentially producing more error-prone code.

UAT is a way of testing new features. Testing is an example of a program analysis technique that aims to check if a program behaves correctly. This technique is executed at run-time (i.e., during program execution) and requires human interaction with the application. This type of testing is often costly to achieve since it requires human resources, and it can be prone to human error as the tester might make some mistakes or not consider all the relevant executions. For example, in the previously mentioned code, suppose that the tester executes only two orders: the first contains only one product, and the second has three products inside. In this case, the program seems

¹<https://262.ecma-international.org/5.1/#sec-10>

to behave correctly, and the bug is undiscovered.

Although other types of testing exist (for example, automated tests executed automatically by a machine), this technique is inaccurate since it is sometimes hard to reproduce an execution.

For a better analysis of the correctness of software, one could integrate a static analysis tool on the CI/CD of the project that scans the source code and finds defects, compliance issues, and vulnerabilities without running the code. For example, using a static analysis tool like SonarQube², the bug in the code 1.1 would have been easily identifiable (figure 3.1).

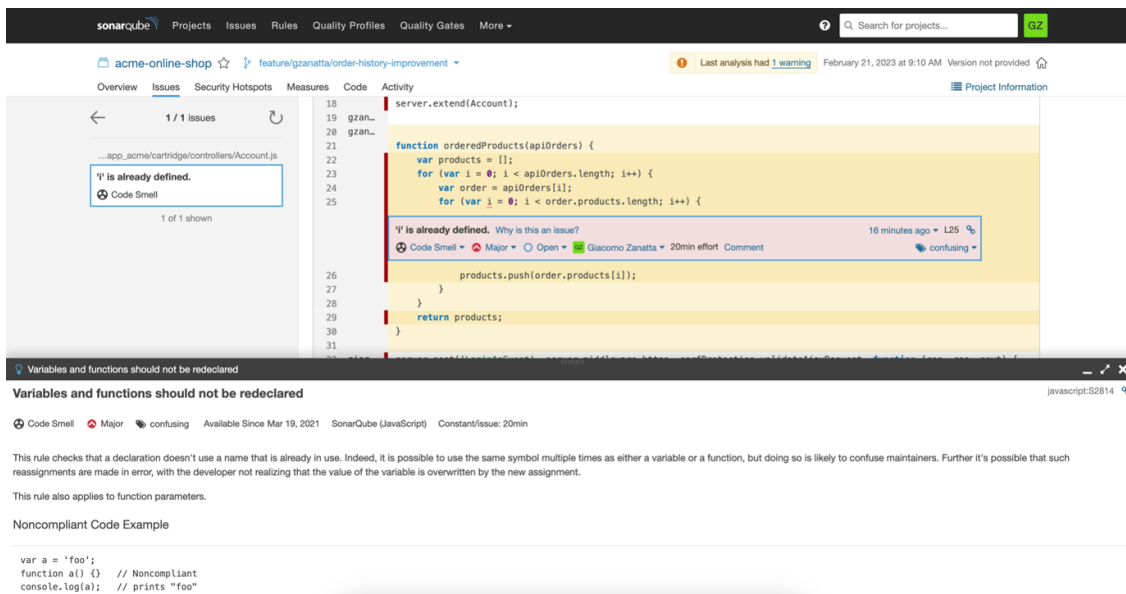


Figure 1.1: SonarQube output for code 1.1

Integrating a static program analysis tool inside the CI process of a project permits the discovery of bugs during the development phase: for example, a static analysis tool can be configured to run at every commit to a repository and blocks the ability to merge a pull request if the analysis fails due to the detection of a bug, providing helpful insight on how to solve them.

This thesis talks about static analysis and, more precisely, about LiSA, a static analysis framework. We will see how LiSA can perform some static analysis of Python code to extract valuable (from a correctness point of view) things from it and learn how we can use this framework to perform domain-

²<https://docs.sonarqube.org/latest/>

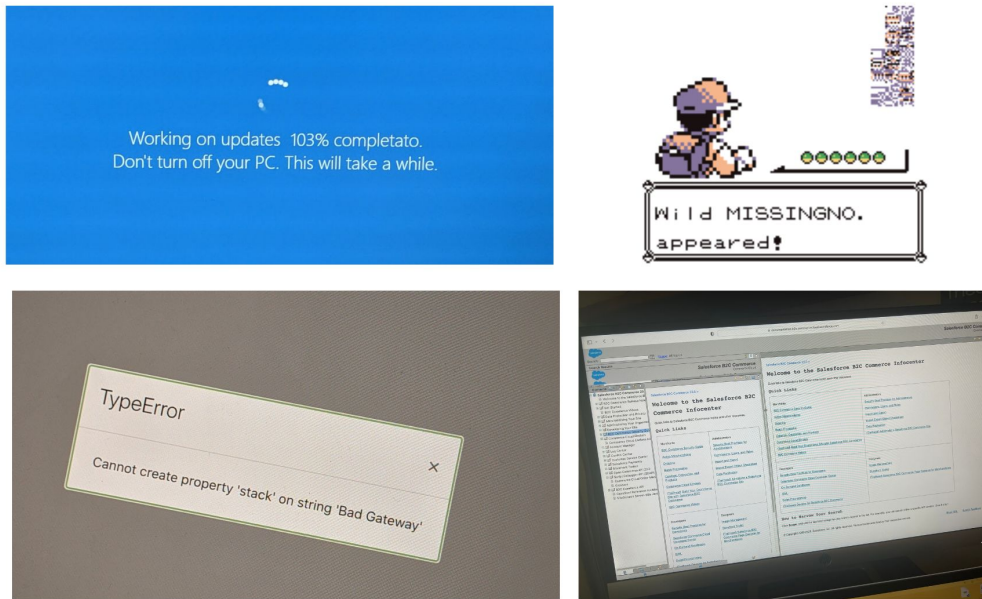


Figure 1.2: Some bugs: an infinite update, the famous MissingNo Pokémon^a, an error on an error message, a recursive documentation.

^a<https://www.kotaku.com.au/2014/11/pokmons-famous-missingno-glitch-explained/>

specific analysis. More concretely, we will perform analyses that are mainly focused on programs with a well-defined application in the real world: programs for robots.

1.1 Bugs and Safety-Critical Systems

Bugs are everywhere. They could hide inside a web application function (like the one we saw before), waiting to be triggered by an unaware user. They could stay inside your smartphone's operating system, causing some funny effects on the graphical user interface, like, for example, showing a negative percentage of battery. They are around us, and if you have a good eye, you will likely find at least a bug a day... Or an hour.

Figure 1.2 shows some typical innocuous bugs. But there are also bugs that can greatly impact the environment where the machine that executes it lives. Take, for example, figure 1.3.

On June 4, 1996, the maiden flight of the Ariane 5 (a rocket of the ESA,



Figure 1.3: The explosion of the Ariane 5 flight, June 4, 1996

European Space Agency) launcher failed, destroying itself³ at an altitude of 3700 meters. The failure analysis found that the problem happens during the execution of a data conversion from a 64-bit floating point to a 16-bit signed integer value, as the code in 1.2 shows⁴. This conversion causes an unhandled operand error because the number converted had a value greater than what could be represented by a 16-bit signed integer. The bug cost 370 million dollars.

```
1 P_M_DERIVE(T_ALG.E_BH) :=
2   UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *
   G_M_INFO_DERIVE(T_ALG.E_BH)))
```

Code 1.2: The line of code that perform the unsafe casting

If a simple conversion typo could lead to these consequences, imagine what a bug can do in other safety-critical systems: these systems are systems where a software issue could lead to a hardware malfunction, which results in human injury or environmental damage. [39]

The Ariane 5 failure report⁵ says that *“The extensive reviews and tests carried out during the Ariane 5 Development Programme did not include adequate analysis and testing of the inertial reference system or the complete flight control system, which could have detected the potential failure.”*. This statement implicitly tells us about the importance of analysis of programs to detect and address possible hidden bugs.

1.2 Static Analysis for Robotics

And after these considerations, we will talk about the main topic of this thesis: static analysis for robotics.

Static analysis is a special program analysis technique that aims to discover bugs without running the program in the early phase (but not necessary) of the DevOps life cycle (the blue arrows in figure 1.4). But why are we going to use static analysis for this thesis? Looking at figure 1.5, we see that most bugs are introduced in the early development phase. Notwithstanding, we have a low detected error rate in these phases (Analysis, Conceptual Design,

³https://www.youtube.com/watch?v=gp_D8r-2hwk

⁴https://de.wikipedia.org/wiki/Ariane_V88

⁵<http://sunnyday.mit.edu/nasa-class/Ariane5-report.html>

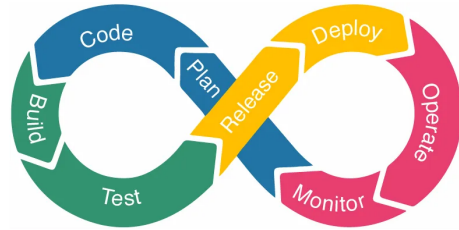


Figure 1.4: DevOps life cycle

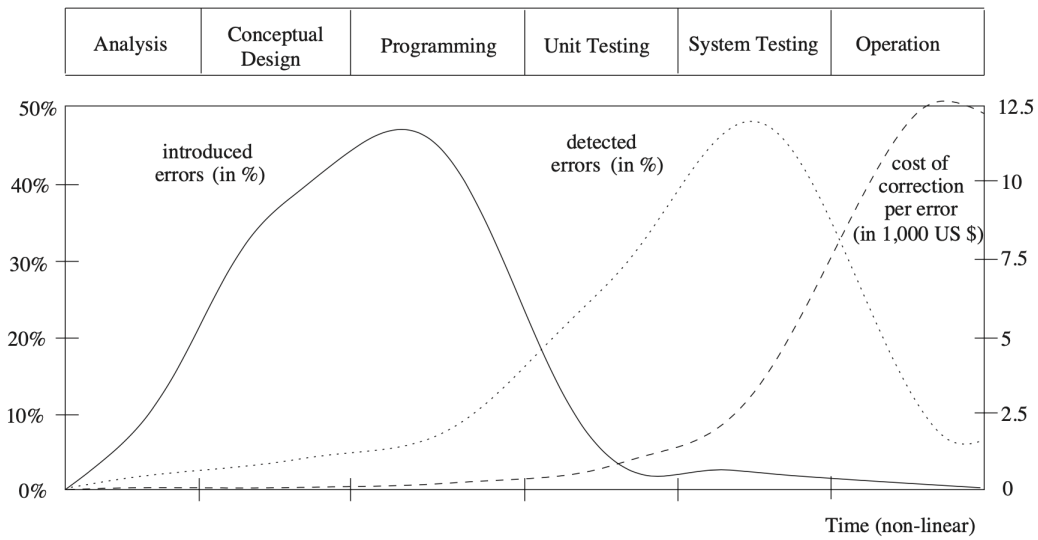


Figure 1.5: Software lifecycle and error introduction, detection, and repair costs [6]

and Programming). But look at the dashed line: the cost of a bug correction is low too! This is because when a bug is detected in a production environment (i.e., the shipped-out and ready-to-use application), correcting the bug means that one needs to figure out what's going on, fix the problem (keeping particular attention that the fix doesn't involve and break any other part of the application), re-execute all the tests and after all that prepare another build to release to the public, hoping that all the users update the software. The effort to fix a bug in a feature that is already released in the production application is huge with respect to fixing it during development.

Talking again about the Ariane 5 rocket explosion, if the conversion bug had been discovered during the software's development phase, the rocket probably would not have exploded (and much money would not have gone up in flames). A static analysis tool can detect such bugs without too much effort. We will focus on the analysis of robotic applications because robots nowadays are more ubiquitous than ever, and some fall under the categorization of safety-critical systems. Since some special robots cooperate with humans in the real world to accomplish dangerous tasks, these entities' behavior must not harm the people around them and the environment. And since the behavior of a robot is defined by the software that is executing, it is important to put particular attention on the correctness of the source code. Static analysis can do this, as we will see in the next chapters, aiming to compute the semantic properties of programs and to find security issues and bugs in general.

In this thesis, we are going to use LiSA [29], a static analysis framework, and we will add the support of the analysis of ROS2 [28] Python programs defining the semantics of the ROS2 python library and analysis on the ROS Computational Graph (these concepts will be explained in the appropriate chapters).

The effort made in this thesis could be a starting point for subsequent analysis of ROS2 programs with LiSA.

1.3 Thesis Structure

Chapter 2 will set the necessary preliminaries for this thesis: programming languages, the semantics of programs, the correctness of software, and static analysis.

In Chapter 3, a static analysis library (LiSA) will be introduced and detailed.

Here will be explained concepts like front end, SARL, and call graph.

Chapter 4 will tell us about ROS (Robot Operating System), a framework for building robot applications, providing an overview of its Python client library and a simple example. In Chapter 5, we will talk about the state of the art of static analysis for robotics, motivating what we will do and why we will do that in the chapter right after.

Chapter 6, the beating heart of this work, will show how to build an analysis for ROS inside LiSA.

Chapter 7 and 8 concludes this thesis.

Chapter 2

Preliminaries

Unfortunately, defining in detail what a computer is and how it behaves requires much time, and it is not the purpose of this thesis. However, in this chapter, we provide a brief high-level overview of this fantastic world to give the reader the basic knowledge necessary to follow this text. Without spoiling too much, in this chapter, we will talk about machine language (i.e., the language a computer talks), programming languages, semantics of programs, program analysis, and static analysis.

2.1 Computers and Machine Language

Computers (also known as machines) are strange electronic creatures that exist with the sole principle of following and executing orders defined in a language that they can understand. This language is called machine language, characterized by an alphabet composed only of 0s and 1s. If you want a machine to do what you want, you must speak its language and write some statements using only 0s and 1s. A sequence of one or more statements defines a program. In other words, a program is a set of instructions a machine executes. Why do computers use this language? To answer this question, we need to step back and analyze how a computer works. Experts will tear out their hair reading the next few lines (and the author hopes they will close an eye, too), but for the sake of simplicity, imagine a room with a light bulb controlled by a switch. When you press the switch, the bulb will turn on, and we will have light. If you press again, the light will go out. The light switch controls the underlying electric circuit: if the switch is OFF, the circuit is open, and no electricity flows. Otherwise, if the switch is ON, the circuit is closed, and electricity will flow without any

problems, reaching our light bulb that will bring brightness to the environment. Now think about millions of these light switches, each of 5 nanometers wired together in a surface of 100mm^2 [23]. These switches are called transistors and are wired to each other so that some of them can turn ON (or OFF) other ones. These transistors live in the CPU (Central Processor Unit), one of the computer's principal components responsible for processing the previously mentioned instructions. Some of them have a special task: for example, some transistors are grouped in a special CPU unit called ALU (Arithmetic-Logic Unit) and perform operations, while others (registers) are arranged and combined, remembering their state during the execution of instructions. Another essential section of the CPU that needs to be mentioned here is the Control Unit (CU), which orchestrates the execution of instructions. An instruction will turn ON or OFF some transistors that propagate their state to others, making them turn on or off based on how they are wired together. An instruction is nothing more than a pattern that says: *"Turn on these transistors, turn off this one and this other one, turn on these..."*. A natural way to represent the state of a transistor is to use 0s and 1s. With 0, we encode the absence of electricity (i.e., an OFF state). With 1, we encode the other state (ON). So an instruction is nothing more than a list of 0 and 1 representing the states of some transistors. The transistors involved in an instruction propagate the state among other transistors until some special transistors are reached. The state of the latter transistors represents the output of the instruction. Using 0s and 1s, you can encode almost anything, and this language is called machine language. For example, this string `0000001011011000101000000100010` for some computers could represent an operation like the subtraction of two numbers.

As a note of mention, a computer can't understand what's going on: it does not have a concept of the program's meaning; it just executes instructions without thinking about it like a loyal subject does the bidding of his own king. A machine cannot understand if the running program (i.e., a program that is executing) is correct.

Please note that this is a very simple example, and computers do not really work like this: for more specific details, we invite the reader to take a look at the book *"Computer Organization and Design: The Hardware/Software Interface"* [32]. But this premise was necessary to introduce programming language.

2.2 Programming Languages

A computer “talks” only with 0s and 1s. It receives a string of 0s and 1s, and after a bit, it produces an output based on what it gets as input. But there is a problem: all form of human language is very different from this language. It’s tough for a human to learn machine code and to write programs using it. To make things easier, programming languages were invented. They are some sort of abstraction that permits the definition of instructions more easily, usually (but not necessarily) using identifiable words. These instructions are then translated into machine code with some techniques (and this is another story) to make them comprehensible to the computer. A set or a sequence of instructions defines a program executed in a specific order by a machine. These instructions stay in one or more files called source codes.

Source codes are made by lines of words defined over an alphabet. Like human languages, programming languages have grammars: a grammar is a set of statements that defines the syntax of the programming languages. To write a program in a language, you must follow the language’s syntax. Syntax describes the rules for how to write a valid program in a language.

Consider, for example, the phrase, “*The cat is on the table.*”. This phrase respects the syntax of the English language, so it is a valid English statement. The same concept applies to programming languages.

2.3 Compilers

A way to transform a programming language into machine code is by using a compiler. A compiler is a program that takes another program written in a specified language and converts it into another language. Within the usage of a compiler, we can write our program in a high-level language (i.e., a language with strong abstraction from the underlying details of the machine), and the compiler will give us the appropriate program ready to be executed in our machine. Note that it is not always like this: for example, there could be an intermediate language between our programming language and the machine language.

For further references about compilers, an interesting book could be “*Compilers - Principle, Techniques, and Tools*” [1]. In the next subsections, we will briefly see the first phases a compiler performs when he translates a program. These concepts will be useful to understand better how LiSA works (Chapter 3).

2.3.1 Lexer

The first phase is called lexical analysis, and a lexer performs it. A lexer scans the source code's characters and groups them into meaningful sequences (lexemes). For each lexeme, it produces a token. A token has an associated identifier that permits reasoning about the lexeme type (and what to do with it) during the next phase.

2.3.2 Parser

Syntax analysis is the next phase. A parser (the entity that has the duty to perform syntax analysis) takes in input all the tokens produced by the lexer and checks their validity from a grammatical point of view. For example, if the grammar says that at the end of a statement that does not start with a specific word is necessary to put a ";", when the parser encounters a statement in the source code that does not follow this rule, then it raise an error since the statement is not in the language, thus it does not know how to handle it. Parser produces a tree structure called syntax tree that shows the relationships between tokens. The syntax tree is a helpful structure concerning the semantic extraction of statements and, more generally, programs.

2.4 Semantics of Program

Before discussing the semantics of programs, an important concept relevant to this thesis, we need to take a step back and define the semantics. In short, semantics is the study of meaning. But the meaning of what? This could be a concept, or a concrete thing, or an abstract one. For example, one could study the meaning of life (like Plato did some years ago), music, or paintings. With some imagination, you can study the meaning of anything. Take, for example, figure [2.1](#):

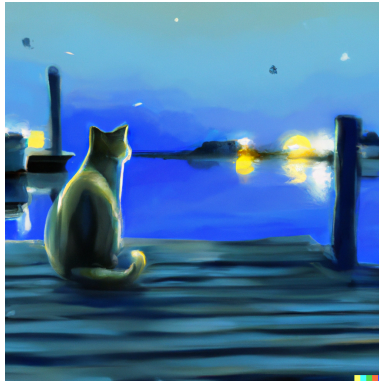


Figure 2.1: A cat sitting on the bay at night (image generated by OpenAI’s DALL·E)

What is the meaning of this image? If you ask this question to four different people, you could have these answers back:

- *“Easy: this image represents a cat.”*
- *“Hey, this is a bay!”*
- *“Well, this is a graphic meaning of the song ‘(Sittin’ On) The Dock Of The Bay’ by Otis Redding. ^[1]”*
- *“This image means ‘waiting’.”*

Note that every answer is correct. The interviewers simply focused on different points of interest in the same image, respectively: subjects (a cat), environments (a bay), feeling (the music), and actions (waiting). These points of interest are the semantic properties of what we are studying.

After this brief introduction, we can talk about the semantics of programs. As you can imagine, the semantics of a program is its meaning. Xavier and Kwangkeun, in the book *“Introduction to Static Analysis”* [43], define the semantics of a program as the description of how it behaves when it is running on a machine. And since (as we said before) computers execute software as they are written, the run-time behavior of a program is solely defined by the meaning of its source language.

A semantic property of a program is a special characteristic of its behavior. Some examples of semantic properties are:

- The fact that a program terminates.

¹<https://www.youtube.com/watch?v=rTVjnBo96Ug>

- The presence of a null pointer.
- The presence of security vulnerabilities like buffer overflows or SQL injections.
- The presence of dead code (code that is never executed).
- The presence of code redundancy (i.e., duplicated code blocks).

2.5 Program Analysis

Techniques that check if a program satisfies a semantic property are called program analysis, and a program analysis tool is an implementation of program analysis. The satisfaction checking of a semantic property is not always trivial: some properties are not computable, like the program termination check. For the latter case, if it exists a program analysis tool that can say automatically and in a finite time if a program terminates, then the “*Entscheidungsproblem*” [26] (i.e., decision problem) will be solved [41].

2.5.1 About Completeness and Soundness

The decision problem mentioned above leads to some considerations. Since semantics properties can be defined from a program’s executions, any non-trivial semantic properties are not computable, as Rice’s Theorem says.

Theorem 2.1 (Rice’s Theorem). Let L be a Turing-Complete language, and let P be a non-trivial semantic property of programs of L . There exists no algorithm such that, for every program $p \in L$, it returns *true* if and only if p satisfies the semantic property P .

What can we do in these cases? There are various methods available to conduct effective program analysis for these non-trivial properties. To circumnavigate the problem, one could limit the automation by requiring users to provide information about the analysis. This approach could work, but we need to remind that this process may be error-prone, and for large programs, it may be the case that the effort required by the user is huge. Another technique consists of relaxing the conditions of program analysis. Instead of giving up on computation, we could let the analysis return sometimes inaccurate results. In this case, if the analysis is unsure about the presence or not of a property, it could return “*I don’t know.*”. This approach

consists of computing an approximation of the program. The inaccuracy comes from the fact that you are losing precision by approximating something, but it does not mean that the analysis is wrong.

We could have two types of approximations:

- **Soundness** We say that a program analyzer a is sound with respect to property P if, for any program $p \in L$, $a(p) = true$ implies that p satisfies the property P .
- **Completeness** We say that a program analyzer a is complete with respect to property P if, for every program $p \in L$ such that p satisfies P , $a(p) = true$.

Completeness and soundness are dual properties. A sound analysis will reject all programs that do not satisfy P but also some that indeed respect P . A complete analysis instead will accept every program that satisfies property P , but also some of them that do not respect the property.

It could be wonderful to have both a complete and sound analysis. Such analysis would be perfectly accurate and would neither miss a property nor return a wrong or inaccurate result. However, due to the theorem mentioned above, such analysis is impossible to obtain: we can't achieve an analysis that can compute a program's non-trivial property in a fully automatic, complete, and sound way.

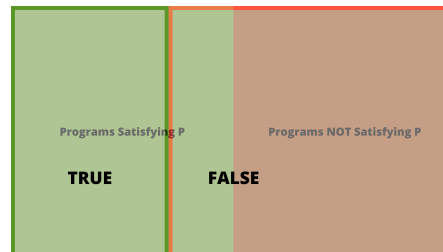
Let's see the diagrams in [2.2](#) to better understand soundness and completeness. The outer rectangle represents a set of programs. We classify these programs into two categories: programs that satisfy a property P (the inner light green box) and programs that do not satisfy the same property (the inner orange box).

Figure b shows how a sound and incomplete analysis behave. For the property of interest P , a sound and incomplete analysis will answer 'no' (false) (the shadowed box with red border) at the question "Does the program satisfy the property P ?" for all the programs that effectively do not have the property, but it could be wrong and answering no also for some programs on the light green box.

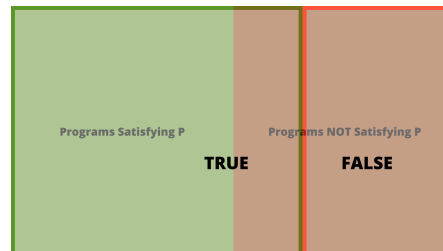
Figure c instead shows the behavior of a complete but unsound analysis. If a complete analysis returns true, then the program has the property. However, for some programs of the orange box, the analysis will produce a wrong result (true instead of false).



(a) Set of programs



(b) Sound but incomplete analysis



(c) Complete but unsound analysis

Figure 2.2: Soundness and completeness

2.5.2 Program Analysis Techniques

There exist different techniques to approach program analysis. For example, we could have some techniques that aim to be automatic, other ones that permit us to compute complete results, and others that focus on the soundness part. Let's see some of them.

Testing

One technique is called testing. This technique has the scope to understand the correctness of the behavior of a program by observing a finite set of finite program executions. It is a dynamic technique since it is performed during the execution of the software.

Although this technique is powerful, it is often costly, and it is hard to achieve proper coverage of executions in large and complex programs.

Developers could write specific code that permits them to run a program use case and check if the output is correct. These tests are typically performed by developers locally or integrated into a Continuous Delivery / Continuous Integration (CD/CI) pipeline to enhance the software's overall quality. For example, a DevOps engineer could put a rule that if a merge request fails a test or has a test coverage below a certain threshold, then the request cannot be merged.

There exist different types of testing: we have automated testing (performed in an automated way like the one we mentioned just a phrase ago) and some testing that requires a human to perform it. An example of the last case could be, for example, testing the checkout process of an e-commerce system directly from the development instance's website. The latter case can be useful to find if an order is placed correctly and if the content presented (i.e., images, translations, popup messages) is shown right with regard to the prototype, just simulating what a final user does on the website.

It is important to mention also Software Integration Testing (SIT), which is a specific typology of testing used when dealing with complex programs, especially the ones that are composed of more than one heterogeneous platform (like, for example, the e-commerce system just mentioned, where at some point it is required that the order placed by a user leaves the system to get visibility of it in the Order Management System (OMS), an entity external to the e-commerce platform, that post-processes orders and handle warehouse stocks and shipments). When the business requires changes that impact more than one system, a SIT is usually performed, which requires synchro-

nization between the organizations that manage the involved systems. The test of a change in the export flow of orders, just to continue the proposed example, starts from one system (e-commerce) and ends in another (the OMS or the Customer Relationship Management system, as known as CRM). Testing has the characteristics of being unsound and complete.

Model Checking

Another technique is the so-called model checking, which consists of checking a property's satisfaction in a program model. This technique puts the focus on finite systems (systems whose behavior can be enumerated) and is based on models describing the possible system behavior in a mathematically precise and unambiguous manner[6].

Typically this approach consists of some phases:

- **Modeling phase:** in this phase, the system is modeled according to the description language of the model checker. The properties to be checked are formalized using a property specification language. Usually, a model is expressed using finite-state automata, with states that define a description of the system's status in a given moment and transitions that say how the system evolves from one state to another. The property specification language is based on temporal logic that permits to model the behavior of systems over time. In this phase, a validation of the model must be performed to check if the formalized problem statement is an adequate description of the actual verification problem.
- **Running phase:** in this phase, we run the model: here, we use an algorithmic approach to check the validation of the considered properties in all model states.
- **Analysis phase:** the last phase consists of an analysis of the checker's results. We could have three different possible outcomes:
 1. The property is valid. In this case, we analyze the following property, or, if it is the last, the model is concluded to possess all desired properties.
 2. The property is not valid. When the analysis output a negative result, there could be a modeling, a design, or a property error. In the first and second cases, the model must be corrected, and

the verification must be redone. Since we are changing a model, all previously checked properties must be re-analyzed. In the latter case, the property may not respect the informal requirement that had to be validated. To handle this, we need to rewrite the property specification and perform a new validation of the model. Since we are not performing modification on the model side, it is not required to relaunch the analysis to check all the properties already analyzed.

3. The model is too large. Sometimes it could be the case that we have a so-called explosion of the state space (too many states to handle in memory). Researchers have found some ways to address this problem. For example, to reduce the state space, we could use symbolic techniques like Binary Decision Diagrams (BDD) or partial order reductions [11].

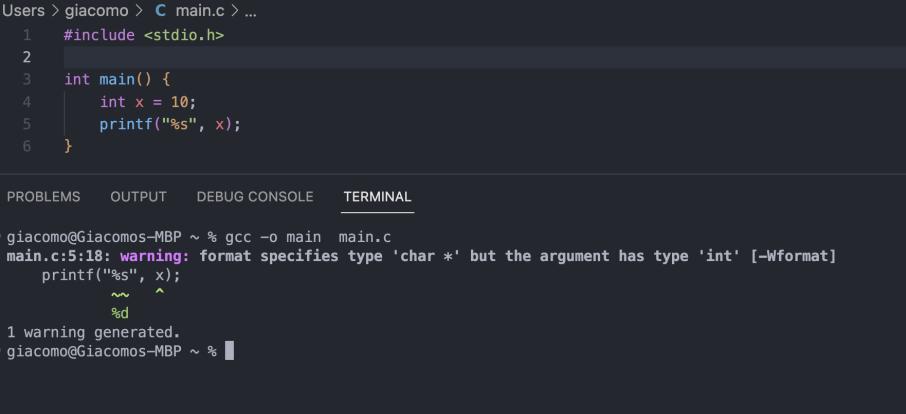
Model checking is an automatic, sound, and incomplete technique with respect to the input program. Although it works on a model of the program and not directly on the program itself, it can be used to verify some interesting properties like the ones proposed in [8]. Still, it is prone to the state explosion problem, as previously mentioned, especially when dealing with the asynchronous composition of processes.

Static Analysis

Another approach is static analysis. It permits the computation of conservative descriptions of program behaviors using finite resources, over approximating the set of all program behaviors in finite time using a specific set of properties. This approach has a dedicated section in this chapter. For now, we just say that static analyzers are sound but incomplete (i.e., they cannot represent all program properties and rely on techniques that enforce termination of the analysis).

The techniques mentioned here have different approaches to program analysis. Testing is a dynamic technique performed by running the actual program. On the other hand, static analysis and model checking use a static approach, but the former focuses on the program itself, whereas the latter concentrates on a finite model of the program.

When talking about software verification in general, there isn't a technique better than the others. Some properties can be easily checked by testing,



```
Users > giacommo > C main.c > ...
1  #include <stdio.h>
2
3  int main() {
4      int x = 10;
5      printf("%s", x);
6  }

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

giacommo@Giacomos-MBP ~ % gcc -o main main.c
main.c:5:18: warning: format specifies type 'char *' but the argument has type 'int' [-Wformat]
    printf("%s", x);
           ~~~ ^
           %d
1 warning generated.
giacommo@Giacomos-MBP ~ %
```

Figure 2.3: The gcc compiler’s type checker at work: there is a mismatch between the specification of the format to print and the type of the argument.

while others can be by using static analysis or static analysis. To improve the overall assessment and quality of software, it is often the case to adopt a combination of these techniques instead of just one. This is typically done by exploiting the CD/CI pipeline, introducing intermediary steps on the deployment that perform, for example, a generic static analysis assessment of the quality of the code and an automatic testing phase.

2.6 Static Analysis

As we said, static analysis is a program analysis technique. Static analysis can help improve the overall quality of a program: from a very high-level perspective, this technique tries to find problems and errors just by looking at the program’s source code. It is a generic term. It could be seen as a toolbox full of different program analysis tools, each with a specific purpose. If we put a hand inside this toolbox, we could extract a checker [1], which is a tool able to find errors and mismatches about a program’s semantics (and syntax). A type checker can, for example, raise a hand and say, “Hey, you are adding an array with an integer!” when it finds that an operand is applied to invalid operators. Another example of a checker is the uniqueness check: sometimes, it could be the case that something must be defined exactly once. This type of checker checks if there is a duplicate of something inside the code that must be unique (for example, in Pascal language, labels in a case statement must be distinct).

Looking back at the content of the toolbox, we can find tools used to perform code optimization: talking about compilers, there exists a subset of them (called optimizing compilers) that tries to minimize (or maximize) some dynamic attributes of the program that is compiling (for example the running time or the memory footprint) just by looking at the semantics of the source code and tweaking some things. The code improvement phase of a compiler generally includes a control-flow analysis and a data-flow analysis. Control-flow analysis checks control-flow graphs of a program to determine path invariant facts about points in a program [38]. The control-flow analysis permits optimizing a program by, for example, substituting function calls with inline functions when possible. Going further, there are some cases where the overall control-flow graph of the program doesn't have a well-defined shape: this happens for programming languages that have dynamic dispatch, where which procedure or function invoked depends on run-time values. In this case, finding the correct call to perform is not trivial, and we will see how a static analyzer can accomplish this task in the next chapter when we talk about the call graph constructor [25].

Control-flow analysis goes hand in hand with data-flow analysis [31], a generic static analysis that permits reason about how the variables are defined and used in the program.

Some application examples are constant propagation, which finds and evaluates constant expressions at static-time (substituting the expressions with a constant to avoid unnecessary evaluation at run-time), and reaching definitions, used to identify all possible assignments to a variable and then find which of these assignments will be effective at each point in the execution of the program.

It is important to say that compilers are not wizards, and some optimization can be exploited if the code is well written. For example, if you are writing a program in C that handle a large dataset, you can help the compiler to optimize the program by writing a cache-oblivious algorithm [22]. But this is another story.

With static analysis, we are going to verify a specification. This means that we need to use mathematical proofs that a program's semantics satisfies a specification. With specification, we intend a property of the program semantics, considering the uncomputability problem mentioned a section ago. A technique used by some static analysis tools to deal with the uncomputability of non-trivial property is to finitely over-approximate a set of all program behaviors using a specific set of properties, the computation of which can be

automated. This technique is called Abstract Interpretation [15] [14].

2.6.1 Abstract Interpretation

Abstract Interpretation is a fascinating theory. As we said, the main idea is to over-approximate all program behaviors. This approximation permits the computation of incomplete but sound results and answers questions about some properties that do not require full knowledge of program execution. In this case, we let the analysis provide, sometimes, inaccurate results. The key is to use some sort of abstraction and compute the analysis on that. An approximation is necessary because the concrete universe of program executions could be infinite (we could have infinite executions for a program).

An abstraction can be seen as a map function between concrete values into abstract ones. Consider, for example, integers. An abstract representation of an integer could be the sign of the integer, and sign is a finite abstract domain. This abstract domain is then a finite approximation of the concrete integer domain.

How, then, can we construct the Sign abstract domain? Well, firstly, we need to figure out what elements of this domain are.

As a starting point, we have the '+' element, which represents all the positive numbers; the '-' element, which represents all the negative numbers; and the '0' element, which represents a number that is neither positive nor negative. But this is not enough, and we will see why.

Suppose we have a function that takes in input two integers and returns the sum of these two. We are interested to study the possible outcome of the returned value's sign. Instead of taking into consideration all the possible integers and saying, for example, "Ok, the first integer is a 1, the second is a 30, the sum of them is 31, and it is a positive number." or "The first integer is -10, the second one is 30, the sum is 20, and 20 is a positive number" we can work just by looking at the sign of the number: all positive numbers will go into the '+' element of the abstract domain, vice versa for the negative number. The number 0 is contained in the '0' abstract value. So we could say, "The first integer is a positive number, also the second, then the sum is a positive number." or "The first integer is a negative number, also the second, then the sum is a negative number.". Here we are losing precision since we are performing an approximation (we abstract away the concrete value of the number, and we take only its sign). But what happens when we have the first number with a negative sign and the second with a positive

one? The outcome will depend. About what? About the concrete value of the integers. If the first number is -1 and the second is 2, the output is a positive number. If the first number is -10 and the second is 5, the result is a negative number. Or, if the first number is the second negated, the outcome is zero. But since we are dealing with the Sign abstract domain, we can't consider the concrete value of the integers! In this case, we can say that we don't know precisely the sign of the result and that it could be a positive, a negative, or a zero. So we add another element on the abstract domain: ' \top ', which represents all the positive numbers, all the negative ones, and 0. With some imagination, this element can be seen as the union of '+', '-', and '0'.

2.6.2 Lattice and Partial Orders

Abstract Domains are lattices [7], which are special algebraic structures based on order theory and abstract algebra. A lattice is a partially ordered set, where every pair of this set has a unique supremum and a unique infimum. A partial order permits the definition of an order to sets that may not have a natural one. We use the notation $a \preceq b$ to denote that element a is less or equal to element b with respect to a partial order. A partial order is a relation between two elements of a set.

A partial order \preceq , together with a set S , defines a partial order set that is written as (\preceq, S) . Consider the Sign domain mentioned above. To define a partial order among the elements, we need to perform some considerations. With what we have said so far, this set has three elements: '-', '+', and ' \top '.

We can define a partial order saying that $a \preceq b$ if b contains all the elements of a . So we have that $0 \preceq \top$, $+$ \preceq \top , and $- \preceq \top$.

Note that it is not required that $a \preceq b$ or $b \preceq a$ for every a, b of the set. If this condition holds, then the relation is a total order.

But is the Sign domain, with the just mentioned relation, a lattice? Short answer, no: we miss defining something. Lattice requires every set pair to have a unique supremum and infimum. The supremum is the so-called least upper bound, while the infimum is the greatest lower bound. Let's see the definitions:

- **Upper Bound:** let (S, \preceq) be a poset and let $A \subseteq S$. If u is an element of S such that $a \preceq u$ for all $a \in A$, the u is an upper bound of A .
- **Least Upper Bound (lub):** an element x that is an upper bound on a subset A and that is less than all other upper bounds on A is called the

least upper bound on A .

- **Lower Bound:** let (S, \preceq) be a poset and let $A \subseteq S$. If c is an element of S such that $c \preceq a$ for all $a \in A$, the c is a lower bound of A .
- **Greatest Lower Bound (glb):** an element x that is a lower bound on a subset A and that is greater than all other lower bounds on A is called the greatest lower bound on A .

Has every pair of the set a lub? Yes, and it is the element T .

Has every pair of the set a glb? No. To answer yes to this question, we need to introduce another element on the set of Signs: the empty set. We call this element \perp . This element is the infimum element of the set and represents an empty set: all the concrete values that are not a number are mapped into this element of the Sign domain lattice.

A lattice can be represented as a Hasse diagram: figure 2.4 shows a graphical representation of the Sign domain. From this figure, we can say at a glance that the glb between 0 and + is \perp , and also for $-$ and $+$. If we consider \top and $+$, the glb is $+$.

So an abstract domain is a domain that over-approximately captures relevant program properties. They are used to perform, for example, numerical analysis [16], heap analysis [20] [18], and information flow analysis [24].

As we said in Section 2.3.2, a syntax analysis aims to build a syntax tree. From a syntax tree, we can extract the semantics of the program's statements. From an abstract perspective, a statement can be seen as a function that takes in input a state of the program and produces in output another state that holds the statement's effect on the program's overall behavior. Generally, the input state is called pre-state, while the output state is called post-state. Reasoning in terms of static analysis, a state models some abstract information about the execution of the program, for example, the content and the structure of the memory at a given program point or the abstract value of an attribute. Considering, for instance, the expression $x = 3 + 5$. The semantics of it says that we are adding two numbers (3 and 5) and storing the results in variable x . Considering the Sign Abstract Domain: in the post-state, we would have, among all the other things previously computed, the information that x is a positive number. But since statements are functions, a problem arises when we encounter a loop.

Suppose having a while loop. We consider the body of the while loop as a

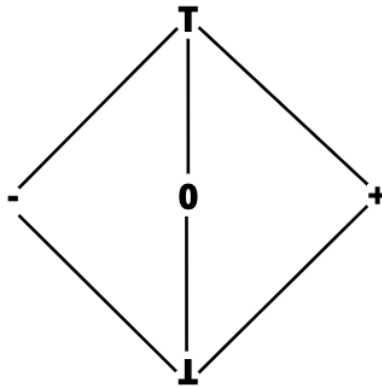


Figure 2.4: The Hasse diagram of the Sign domain

function f . This function is called at every iteration of the loop. It takes in input a pre-state (S_{pre}) and computes a post-state (S_{post}), the latter holding the effects of the current iteration. It is logical and true to think that the S_{post} will become the S_{pre} of the next iteration of the loop. However, it is not always clear how many iterations this loop could perform, and since we want to ensure the computability of the analysis, we need to figure out how to handle these cases: the idea is to look at the S_{pre} and the S_{post} at every iteration. At some point, we would have that applying the function f over a state S it produces in the output the same state S . In this case, we say that S (an instance of an abstract domain) is a fixpoint of the while loop f : if we continue to calculate the value of f starting from S , we will always have S . The main idea is to stop iterating when a fixpoint is reached since the analysis will not change anymore.

Chapter 3

LiSA

LiSA [29] stands for Library for Static Analysis and, as the name suggests, is a Library that easily permits the development of static analyzers. In this chapter, we will analyze how LiSA works and is made.

3.1 Introduction to LiSA

LiSA (Library for Static Analysis) is a tool made and maintained by the *Software and System Verification (SSV)* group at Ca' Foscari University^{1,2}.

As the acronym says, LiSA is a static analysis software library that aims to simplify the development and implementation of static analyzers. The source code, written in Java, is available under the MIT license on Git Hub.

Lisa's engine relies on abstract interpretation theory, providing built-in and extensible classes implementing the main concepts of this technique, such as lattice, control-flow graphs, and abstract domains. How these concepts are implemented will be discussed in detail in subsequent sections.

The main characteristics of LiSA are:

1. **Target Programs:** LiSA provides a framework for building non-domain-specific analysis: this means that the design of the analyses is not focused on a specified family of target language programs. However, thanks to the extensibility of the library, if there is a need to capture characteristics of specified typologies of programs (firmware programs, operating systems, embedded software, to cite some examples), one could develop more domain-specific analyses.

¹<https://github.com/UniVE-SSV>

²<https://ssv.dais.unive.it/>

2. **Source code handling:** LiSA takes in input the program's source code. Program analysis tools with this characteristic fall under the program-level analyses class. This requires implementing a front end that parses the source file to build a program's control-flow graph.
3. **Multi-language analyses:** as just mentioned, a front end is necessary to build the internal CFG used by LiSA to perform the analysis. Since LiSA focuses on extensibility, writing a front end for a programming language not already supported by the library is possible.
4. **Abstraction from the theory:** the underlying theory of abstract interpretation (for example, the fixpoint computation on the control-flow graph) is already implemented in the library. This permits us to build analyses without worrying and focusing too much on the mathematical aspects of static analysis.

Before going further, a disclaimer is necessary: LiSA is a young project still in beta version. This means that the state of the art of the code could change significantly due to the introduction of classes and interface refactoring that leads to breaking changes concerning projects that depend on it. At the date of writing, the last available LiSA version is beta 8 (v0.1b8), while beta 9 is currently under development. The results of this thesis were obtained based on beta 8.

3.2 Project Structure

In this section, we will take a glimpse of the structure of this project. As we say, code is available as a GitHub repository. Navigating the repo, we see four main folders: `lisa-sdk`, `lisa-core`, `lisa-analysis`, and `lisa-imp`. These folders contain the main modules that define LiSA. Let's dive into it.

3.2.1 LiSA Submodules

`lisa-sdk` The `lisa-sdk` module can be seen as the skeleton of the whole infrastructure. It contains all the main classes and interfaces that model the concepts of abstract interpretation. As the name suggests, this is the most important module of the library, and then it requires an additional dissection. Inside `lisa-sdk`, a developer can find these packages:

it.unive.lisa.analysis contains all stuff related to the implementation and definition of abstract domains: here, we can find the implementations of lattices among interfaces defining heap, value, and type domains. Important to mention is the `AbstractState` interface, which models the states of a program, keeping track of variables, memory layout, and memory locations using the domains mentioned above for reasoning about run-time types. An implementation of this interface is the `AnalysisState` class, also shipped out inside this package.

it.unive.lisa.checks this package is the home of syntactic and semantic checks interfaces among their own `CheckTool`.

A syntactic check performs exploitation of the fed program's syntax and permits one to find helpful information about what is written in the code. For example, one could implement the `SyntacticCheck` interface, which is capable of finding all variables and functions with a particular name and recording them thanks to its faithful `CheckTool`. On the other hand, the `SemanticCheck` interface can also have information about the semantics of the program, giving more knowledge of what's going on. This permits capturing details that a simple `SyntacticCheck` could not handle.

it.unive.lisa.interprocedural here, we have the definition and implementation of call graphs and interprocedural analyses. We will talk about it later in this chapter.

it.unive.lisa.logging contains internal classes used for logging purpose.

it.unive.lisa.outputs this package provides the logic that permits LiSA to provide human-readable outputs to the user. Lisa beta 8 can dump the analysis results in JSON and graphviz files instead of interactive HTML pages.

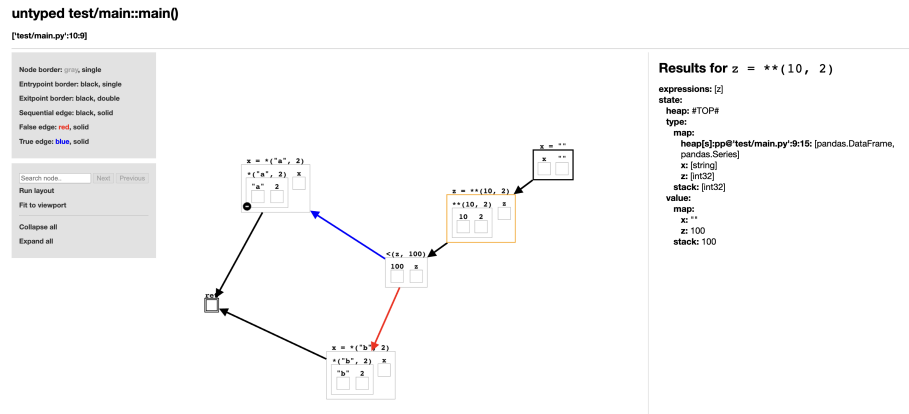


Figure 3.1: Html output of a LiSA analysis

it.unive.lisa.program contains the structure of a LiSA program. A LiSA program is nothing more than a set of control-flow graphs. Although we will talk about it later, nodes of the CFG are statements (that is to say, expressions) of the original program modeled to an internal LiSA statement, while an edge is a direct link between two sequential statements.

it.unive.lisa.symbolic contains the symbolic expressions and operators used by the analysis infrastructure. An implementation of `SymbolicExpression` is a rewritten statement that permits reason about its effects in an abstract domain during execution. It holds information about the static type and run-time types of the statement. Operators cause the transformation of one or more `SymbolicExpression`.

it.unive.lisa.type is the package that contains the type hierarchy. There is also an abstract `TypeSystem`, an Object that tracks the available types of a language. Typically the `TypeSystem` is used in a front end to register all possible findable types in a program.

it.unive.lisa.utils this package contains additional classes and functions for supporting the infrastructure. Here stays the implementation of graphs and automaton used under the hood by LiSA for building data types.

it.unive.lisa This is the main package of the module and contains the main LiSA class used for instantiating the analysis with a simple-to-use

configurator that permits to sets what the analysis should compute and how it should behave.

lisa-program This module defines and implements the principal types and statements of a Lisa program. A Lisa program can be generated from a front end using the provided classes and methods of the library. Here we can find, for example, classes that represent numbers, booleans, and strings types, along with some basic and common statements like numeric Addition and Multiplication and String Concatenation, to name a few.

lisa-analysis This module contains some out-of-the-box analysis, such as constant propagation, reaching definitions, and available expressions. In this module, we find some abstract domains concerning strings (Tarsis[30], Bricks[13]), numeric values (sign, parity, interval), and others.

lisa-imp This module contains a front end for the IMP language, plus an extension and implementation of classes and interfaces of LiSA for handling language-specific constructs, expressions, and types.

IMP is a lightweight and minimal version of Java that inherit from the latter the concepts of Object, throwing away some complex (from an analysis point of view) lexica such as the meaning of access modifiers, interfaces, and static typing.

The main purpose of this module is for internal testing and for showcasing how the infrastructure works.

Submodules Dependencies

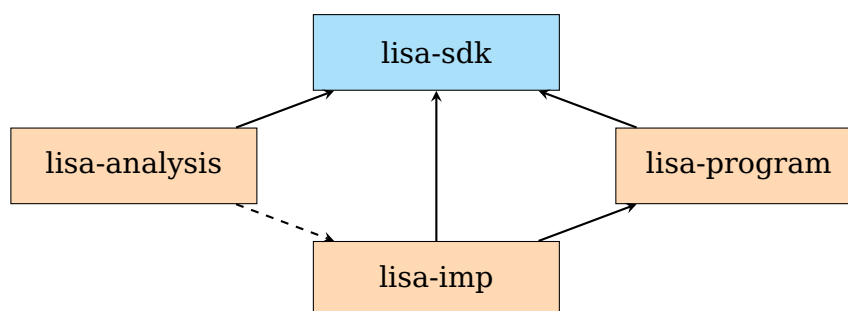


Figure 3.2: LiSA's internal dependencies

Figure [3.2](#) shows how LiSA's internal modules depend on each other. The subsidiary modules are represented in peach, while the main (lisa-core) is in light cyan. A line is a compile dependency, and a dotted line is a test dependency. For example, looking at the figure, we could learn that lisa-analysis modules depend internally on lisa-sdk (i.e., the lisa-sdk module is required to compile lisa-analysis) and on lisa-imp only for building the tests.

3.3 PyLiSA, Front Ends, and Antlr

In this section, we will meet PyLiSA, the Python front end for LiSA that will be our companion from this moment on.

PyLiSA stands on its repository and integrates LiSA modules as external dependencies.

Like its IMP brother, this front end is based on Antlr. But what is Antlr? We will see it in a moment.

3.3.1 Antlr

Remember the definition of parser and lexer that we fix some pages ago? Well. Antlr, short for ANother Tool for Language Recognition, is a lexer and parser generator framework that permits the generation of a parser in a target language for a specified grammar. As previously mentioned, the grammar of a programming language consists of a set of rules that define the language's syntax.

Since writing a parser from zero takes a lot of effort, Antlr comes to help. Suppose that you want to create a LiSA front end for Javascript. Instead of writing your logic for lexical tokens and syntax tree generator, you can study how Antlr works, get the grammar of javascript, generate a javascript parser for Java using Antlr and inject the generated code inside your front end module. What remains to do is to convert the syntax tree generated by Antlr into a LiSA program.

3.3.2 Front Ends and PyLiSA

A LiSA front end must be able to transform a source code into a LiSA program. The way to achieve this is at the discretion of the developer of the front end. The steps that a typical front end performs are:

1. **Lexer phase:** it transforms the source code into a set of tokens, following some rules (for example, rules provided by grammar). This process permits the discovery, along the other things, of language-specific keywords, operators, and symbols.
2. **Parser phase:** this step is able to find relationships among tokens produced in the previous step to build an abstract syntax tree (AST).
3. **Translate phase:** the last step visits the AST in order to create the LiSA program's CFGs (one for every declared method, function, or module encountered). Since LiSA is not aware of the programming language involved in the analysis, it is the front end's duty to extract the semantics of it. This means that the front ends are responsible for defining custom LiSA statements (if the default ones coming out with `lisa-program` and `lisa-sdk` are not suitable for the language they parse) and, of course, the semantics of these statements.

While a framework like Antlr can handle the first two phases, the last step is the most interesting and peculiar, so we could say a few more words about it, but first, let's shake hands with PyLiSA.

PyLiSA, as we said, transforms Python sources into a LiSA program. It can handle Jupiter Notebook too. How can he do it? Take, for example, the next snippet of a notebook (`.ipynb`) file:

```
1 {
2   "metadata": {
3     "kernel_spec": {
4       "name": "python",
5       "display_name": "Python (Pyodide)",
6       "language": "python"
7     },
8     "language_info": {
9       "codemirror_mode": {
10        "name": "python",
11        "version": 3
12      },
13       "file_extension": ".py",
14       "mimetype": "text/x-python",
15       "name": "python",
16       "nbconvert_exporter": "python",
17       "pygments_lexer": "ipython3",
```

```
18     "version": "3.8"
19   }
20 },
21 "nbformat_minor": 5,
22 "nbformat": 4,
23 "cells": [
24   {
25     "cell_type": "markdown",
26     "source": "Test jupyter notebook",
27     "metadata": {},
28     "id": "7ff54803-6b96-4f4b-a671-fff6eb1766a1"
29   },
30   {
31     "cell_type": "code",
32     "source": "x = 3\nprint(x)",
33     "metadata": {},
34     "execution_count": null,
35     "outputs": [],
36     "id": "d9b02c7f-f935-47d0-8cab-36cb60cad033"
37   }
38 ]
39 }
```

Code 3.1: Snippet of a Jupiter notebook file

The figure above shows that a Jupiter Notebook is nothing less than a well-defined JSON file. To work with a notebook, PyLiSA, before lexing the file, reads (and parses) the latter as a JSON. Having the knowledge that code lines stay inside the source attribute of a cell object that has `cell_type` equal to `code`, the front end uses this information to throw away all non-Python stuff (i.e., markdown texts, outputs, metadata, etcetera) and creates the source for Antlr extracting only Python code. After this step, the source obtained is seen as a normal Python code from the analyzer point of view.

The Translation Phase

We will see now how a front end can translate a source code in a LiSA program using the next simple and trivial example:

```
1 if x < 10:
```

```
2      x = 10
```

Code 3.2: Snippet of a Python code

Feeding this program to PyLiSA, Antlr will produce the AST in [3.3](#).

The front end then visits the tree's statements starting from the root, deciding what to do according to the type of encountered statement. Before doing that, since Python doesn't have an explicit definition of the main entry point, PyLiSA creates a new empty CFG that is identifiable. Inside this CFG, it will push the global statements of the program (the statements available when we execute the program).

Looking at the example, the first encountered statement is a `compound_stmt`. A `compound_stmt` is a block of statements, like function definitions, loops (for, while), a try, or an if block.

PyLiSA will then find the type of the `compound_stmt` using the methods provided by Antlr: when it is sure that it is an if statement, then it checks how the block is composed: in other words, it will find answers for questions like: *"Is there an else block? If yes, what it contains? How the test is structured and defined? Where and what is the entry point of the true (false) block?"*.

When PyLiSA analyzes the if statement of the example, it understands that it is dealing with a comparison expression: the comparison is composed of an operator (the < symbol) and two expressions: the first one is a name (x), and the second an integer (10). With these facts, it can infer the semantics (i.e., the meaning) of it and creates a LiSA expression that models the lower-than comparison. It is important to say that the default implementation of LiSA comparison expression can handle only numeric types, while Python language is less racist and can also compare other types such as string and list. In this case, it is necessary to extend the default comparison expressions with language-specific ones: an interested and curious reader can find the extensions of what we are talking about in the package `it.unive.pyliisa.cfg.expression.comparison` of PyLiSA project.

After the creation of a LiSA statement, the front end then adds the statement in the CFG as a node. Then, it will check the true and else blocks of the comparison building the appropriate LiSA statements using the same mentioned logic, creating the edges between them, and it will continue to transform the statements of the AST into LiSA statements.

Now, for better comprehension, it's time to talk more about LiSA's statements and the internal CFG structure.

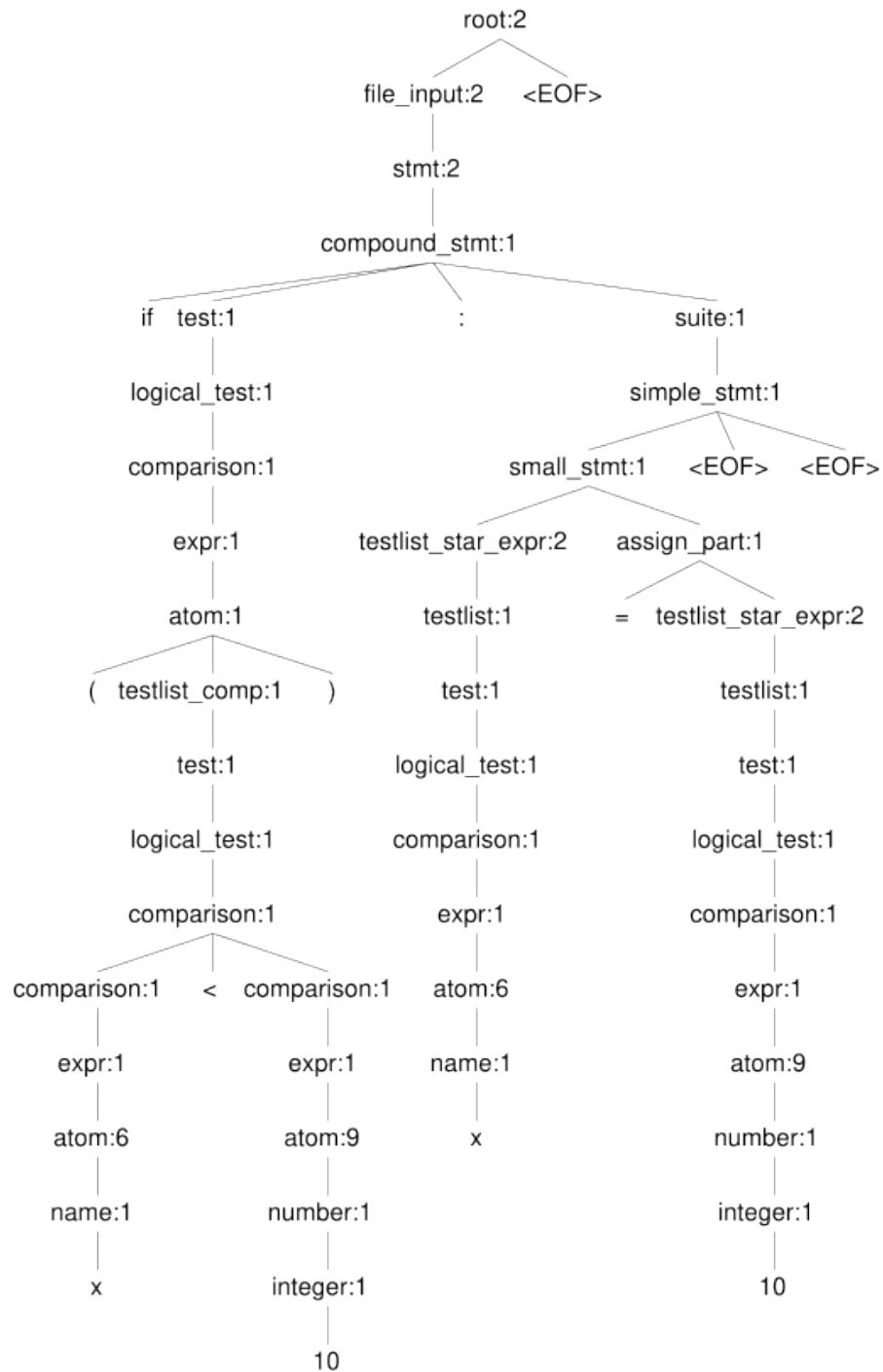


Figure 3.3: The AST obtained by parsing the code in 3.2

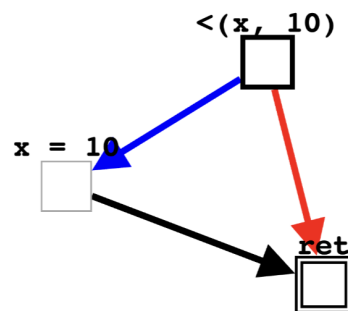


Figure 3.4: The graphical representation of a LiSA CFG obtained from source [3.2](#). Squares are statements (nodes), while arrows represent the linking between statements (edges): the blue (red) arrow is the true (false) edge (the statement pointed by the arrow is the next statement in the flow evaluated only the condition defined in the previous statement is true (false)). Black arrows model sequential edges.

3.4 LiSA's Control Flow Graph

A CFG in LiSA is a structure able to represent a program.

As we said, Statements are nodes of the graph, while edges define how the execution flows along statements. The characteristic of LiSA's CFG is that it uses a flexible philosophy: statements are defined in such a way that language-specific patterns are abstracted away. In order to do so, native constructs are treated as procedure calls (we will see how in a moment).

3.4.1 About Statements

In this subsection, we will provide an overview of the classes and interfaces that models statements in LiSA. In subsequent figures, we define Java interfaces in light cyan and classes in peach.

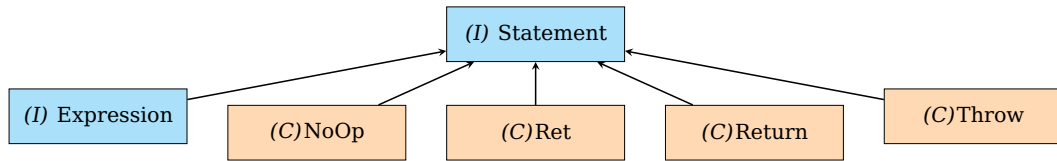


Figure 3.5: First-level inheritance of Statement interface

In figure [3.5](#), direct subclasses of Statement are presented. Here we have four concrete classes and an interface that models Expression. In LiSA, statements that are common to most programming are already defined and implemented due to the fact that they have consistent semantics across different languages. In details:

NoOp NoOp models statements that perform nothing. It can be used for instrumenting branching operations.

Ret Ret models the end of a function that does not return anything to the caller.

Return Return models the ends of a function that returns something to the caller.

Throw models the raising of an error.

Expression Expression is a sub-interface that models expressions. Let's see its main subclasses:

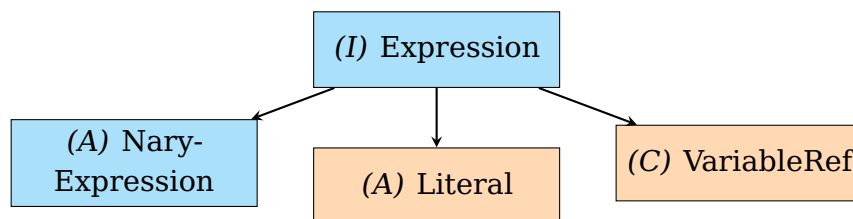


Figure 3.6: First-level inheritance of Expression interface

An expression is one (or more) constant(s), variable(s), function(s), or operator(s) combined together that a program computes in order to produce a value.

Literal Literal is an abstract class that represents a constant value. This class is parameterized with the type of the constant that it represents. Default implementations of this class can be found in the `lisa-program` module.

VariableRef This class models the reference to a variable. A variable is identified by its name.

NaryExpression This abstract class models the composition of expressions (i.e., generic expression with n sub-expressions).

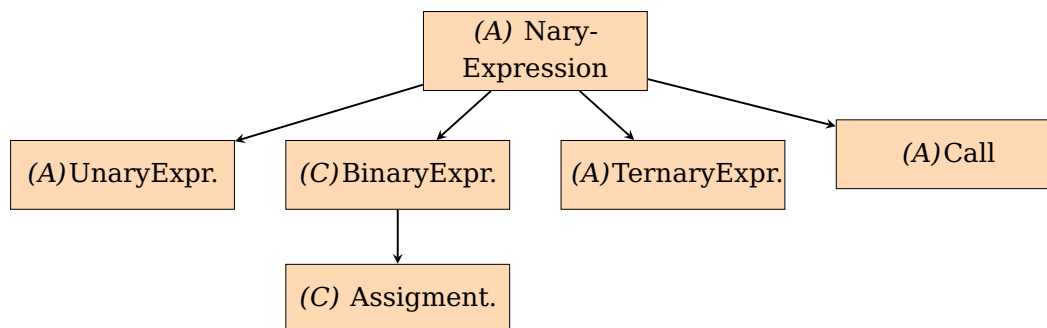


Figure 3.7: Inheritance of Statement interface

UnaryExpression An UnaryExpression is a NaryExpression with a single sub-expression. Two examples of UnaryExpression are the negation of a value ($-x$) and the not operator ($!x$), both of them implemented in `lisa-program`.

BinaryExpression NaryExpressions with exactly two sub-expressions. An example is the assignment ($x=5$) along with addition ($x+5$), Subtraction ($x-5$), Multiplication ($x*5$), and logic constructs involving two values (`and`, `or`), all present in `lisa-program`.

TernaryExpression NaryExpressions with exactly three sub-expressions.

Call Call is an interesting interface that models function calls and, more generically, calls to other CFGs. LiSA defines four main types of calls:

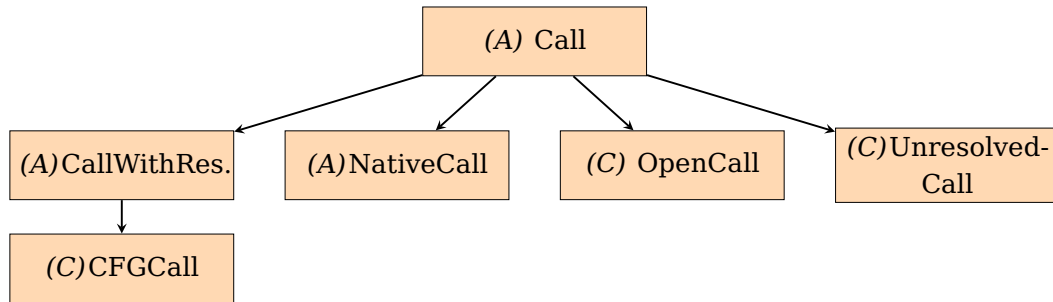


Figure 3.8: The Call classes

NativeCall In this class, native constructs are modeled. Simulating constructs instead of explicitly defining them brings the advantage of having different semantics for the same construct.

OpenCall Calls that do not are submitted to LiSA (for example, function calls to an external library).

UnresolvedCall UnresolvedCalls are calls to a CFG presented to LiSA but not already resolved. UnresolvedCalls are like pointers to some black boxes identified by a name (signature) and a list of parameters.

CallWithResult This class knows how to deal with the call that models it, knowing how to compute its results (i.e., what the call does). CFGCall is a concrete implementation of this abstract class and explicitly defines a call towards one or more of the CFGs of the whole program that is being analyzed.

The resolution of calls is described in detail in the next section.

3.4.2 About Edges

Talking about edges, we have three types of it. An edge, as we already mentioned, models flow between two statements.

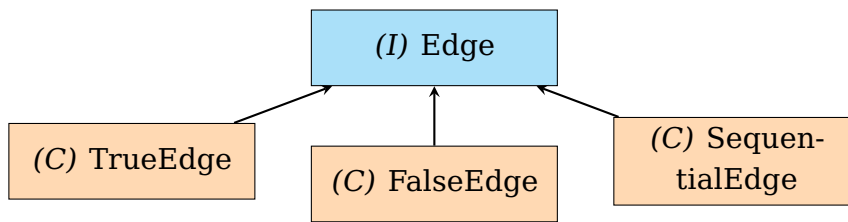


Figure 3.9: Edge classes

TrueEdge This class models a conditional flow: the second statement is executed only if the first holds a true boolean result.

FalseEdge This class, like the above, models a conditional flow, but the second statement is executed only if the first holds a false boolean result.

SequentialEdge This class models a sequential flow. Right after the execution of the first statement, the second will be executed too.

3.4.3 Typing

LiSA's expressions have the capability to model their types.

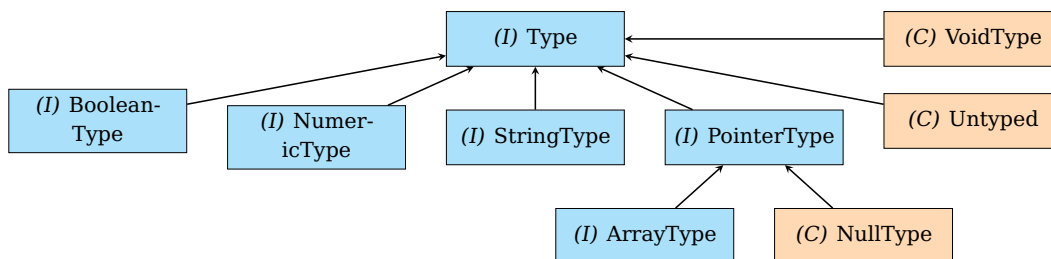


Figure 3.10: LiSA's Type interface

Lisa provides a well-defined hierarchy of types interfaces, with some default implementation like void (used as a return value to functions that does return nothing), untyped (used when we are dealing with non-static-defined variables), and null. The front end must model the types that the referred language accepts in an appropriate way. Although some types are already implemented in the lisa-program module (such as numeric or string types), these provided types could not fit well for all the programming languages. For example, object-oriented programming like Java treats everything as an

Object. To model this peculiarity, standard types implementation (like an integer) should also be treated as pointers to heap location (i.e., the integer type must also implement `PointerType` and not only `NumericType`).

The basic idea of having a type infrastructure strictly defined by the internal type interfaces adds the capability to perform multi-language program analysis (i.e., analysis of programs written in more than one language).

3.4.4 PyLiSA CFG

Some of the main implementations of `Statement` defined in `lisa-program` do not fit well with Python. Take, for example, the `Addition Statement`. The default `lisa-program` implementation of addition considers only numeric addends, while in Python, we can also add together strings or lists, with the effect of concatenation in both cases. PyLiSA handles these cases by extending the default implementation. Furthermore, some Python statements are not implemented directly in LiSA, like the `'in'` and `'is'` expressions or the concept of lambda functions. Some examples of `Statement` extensions can be found in the package `it.unive.pylisa.cfg.statement` of PyLiSA.

3.5 LiSA's Analysis

3.5.1 The Fundamentals

The core of the LiSA's Analysis structure relies on `Lattice` and `SemanticDomain` interfaces for models `Abstract Domains`. These interfaces provide the fundamental of the whole infrastructure, and in this section, we will see the basic implementations.

Before going further, it is important to define also what are the entities targeted by `Abstract Domains`: the symbolic expressions.

SymbolicExpression A `SymbolicExpression` is a rewritten LiSA `Statement`. This transformation is necessary because statements do not have well-defined semantics on their own: consider, for example, the instantiation of a new `Object`. What does this mean intrinsically? This means that a new memory region is allocated, and a pointer to this region is created. Then, the constructor is called, with the parameters and the newly created pointer as a receiver. A statement on its own can't handle such information. Hence, to take care of the meaning of these steps, a statement must rewrite itself

in a symbolic expression: in this way, a symbolic expression can model the side-effect-free expression that is being built on the stack, and an abstract domain can use these to reasoning about the effects that a statement could produce. We have two main categories of symbolic expressions:

1. **ValueExpression**: it models and can deal with constant values and identifiers.
2. **HeapExpression**: it can reason about the heap and the operations that concern it (for example, the allocation of an object or the reference of another one).

Lattice A lattice in LiSA is defined as a generic interface (`Lattice`) parametric to the concrete instance of its implementing class. This permits us to define lattice's logic once for all possible types of elements and to reason about the appropriate returns type of its methods.

Since some lattice operations are common and are not dependent on the specific instance, LiSA has an implementation of the `Lattice` interface (the abstract class `BaseLattice`) that defines these cases for us. The methods that the `Lattice` interface defines are (note that `L` is the concrete class of the `Lattice`):

1. **L lub(L other)**: it returns the least upper bound between the current element (the element for which this method is called) and the element `other` passed as a parameter.
2. **L widening(L other)**: it applies the widening operator between the current element and the element passed as a parameter.
3. **boolean lessOrEqual(L other)**: implements the partial order \preceq . It returns true only if this (the current element) \preceq `other`.
4. **L top()**: returns the top element of the lattice.
5. **L bottom()**: returns the bottom element of the lattice.

`BaseLattice` implements the above methods, handling the common case and delegating detailed computation to the concrete classes that extend it. Some concrete instances of `Lattice` are defined inside LiSA:

1. **SetLattice**: models a lattice where elements are sets. In this case, the least upper bound is implemented as a set union. This class is

parametric to two different classes: S and E, which are, respectively, the concrete SetLattice type and the type of elements that a set can contain.

2. **InverseSetLattice**: like the latter, it models a lattice where elements are sets, but the least upper bound is defined as a set intersection. This class is parametric in the same way as the SetLattice: they are just twins; the only thing that changes is the lub implementation.
3. **FunctionalLattice**: it applies functional lifting to inner Lattice instances that are mapped to the same key. It is parametric to F (concrete class of FunctionalLattice), K (the types of keys), and V (the type of values that must be a subclass of Lattice).

SemanticDomain In LiSA, there is a special interface that models the reasoning of the semantics of statements. This interface is the SemanticDomain, and it is parametric to D (the concrete implementation of it), E (a class that extends SymbolicExpression, i.e., the type that the domain can handle), and I (the type of identifiers that the domain can work with). A SemanticDomain's instance models the abstract information that a variable of the program under analysis holds.

This interface defines the next methods:

1. **D assign(I identifier, E expression)**: it returns a copy of the actual domain, modified with the evaluation of the expression to the identifier.
2. **D smallStepSemantics(E expression)**: it returns a copy of the actual domain, modified with the evaluation of the semantics of expression.
3. **Satisfiability satisfies(E expression)**: Satisfiability is an internal Lattice used to represent boolean values. This method returns the Satisfy value if the expression is satisfied by the current domain (that is, a program state at a given point).
4. **D assume(E expression)**: it assumes that the expression holds and returns a copy of the domain with this assumption.
5. **D forgetIdentifiers(Collection<I> identifiers)**: it forgets all the information about the identifiers.

SemanticDomain has mainly two interfaces that extend it: ValueDomain (with the main purpose of keeping track of semantic properties about program variables) and HeapDomain (that tracks the evolution of dynamic memory during program execution). These two classes permit us to compute the semantics of a symbolic expression, and we will talk about them in a moment: firstly, we want to introduce the Abstract State.

3.5.2 Abstract State

LiSA's Abstract State takes inspiration from [19]. It is an abstract model of variables and heap memory of a program. It keeps track of what's going on in the heap and in the variables of the program under analysis. The underlying models of heap and variable are modeled through two distinct interfaces: HeapDomain and ValueDomain. Since the analysis can be performed only on the variables, the heap must be rewritten in some way in a ValueDomain. This is done internally by the analysis using class-specific methods that we will discuss.

HeapDomain The HeapDomain is a parametric interface that extends Lattice and SemanticDomain. This is the only component of LiSA that knows exactly how an expression is resolved to a memory location. In fact, its main duty is to track how the memory evolves during execution, and this is done by binding its SemanticDomain to HeapExpression. In order to get information about the internal values of the heap, the sub-expression that deals with memory needs to be rewritten. For doing that, this interface has a method, `rewrite`, that, given a SymbolicExpression, returns the set of its ValueExpression, getting rid of all the information about memory and replacing them with heap identifiers.

ValueDomain Like HeapDomain, the ValueDomain is a parametric interface that extends Lattice and SemanticDomain. This interface can deal only with constants, variables, and operators. The inherited SemanticDomain is bound to ValueExpression (i.e., it accepts only expressions that does not concerns with memory). The ValueDomain can also track properties of the heap identifiers produced by an HeapDomain.

An AbstractState implements ValueDomain and HeapDomain. It implements the Semantic operations calling the corresponding operation on the

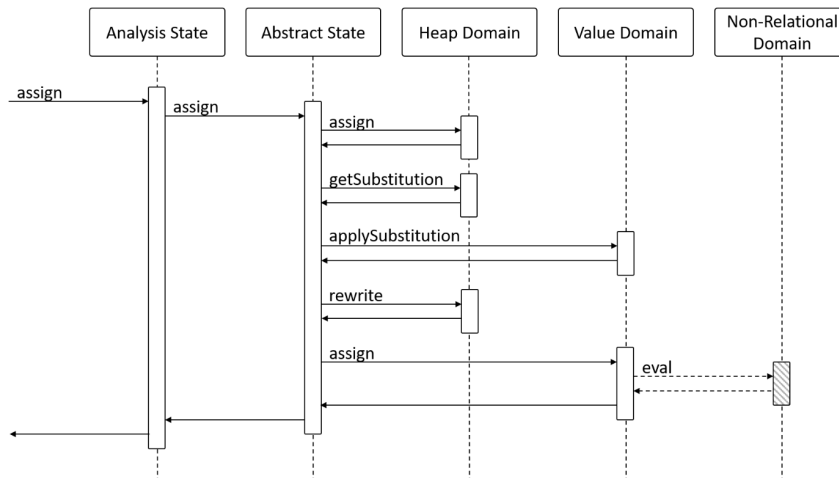


Figure 3.11: Sequence Diagram of Analysis State's assign. Figure taken from [29]

heap domain first and, secondly, the operation on the value domain using also the expressions with heap identifiers produced by HeapDomain.

3.5.3 Analysis State

In order to get a meaningful result, we need some additional information: the program state and the execution state. These are provided by the AnalysisState class, which extends the AbstractState. It is composed of the AbstractState itself, for modeling the abstract values of program variables and heap locations, and a set of SymbolicExpression: the latter permits keeping track of what has been evaluated in a given moment and what is available for later computation. The AnalysisState works in a very fashioned way: we now show as an example how the AnalysisState computes an assignment.

1. When we call the assign method of the AnalysisState, the call is forwarded to the inner AbstractState.
2. The AbstractState calls the assign method of its HeapDomain to compute the assignment's effect in the dynamic memory.
3. An assignment operator on the heap changes the dynamic memory, causing, for example, the creation of heap identifiers or a merge of

them. The `AbstractState` needs to fetch the substitution from the `HeapDomain`. This substitution tells how variables from the pre-state (the state before the assignment) are changed in the post-state (the state after the assignment). A substitution is a replacement of variables.

4. The `AbstractState` applies the substitution on the `ValueDomain` to update it.
5. A rewrite operation is then called in the `HeapDomain`: this method transforms the right-hand of the assignment (for example, a field access) memory-free (with heap identifiers).
6. The rewritten expression is then used to update the `ValueDomain`, calling its `assign` method.
7. At the end, a fresh new `AnalysisState`, with the updated `ValueDomain` and `HeapDomain`, is built and returned to the caller.

These steps are well represented in Figure 3.11. Note that the same process happens for the other semantics operations.

3.5.4 The CallGraph

One question could arise at this point: what about function calls? We talked about only `SymbolicExpression`, heap, and concrete values for now. It is time to talk about calling expressions.

Calls are modeled through the `CallGraph`, an interface that permits the resolution of calls and their abstract result evaluation [3].

Theoretically, a call graph is a special graph that is an abstraction of a program's method calls where nodes are methods and edges are calls. These graphs are flow insensitive, i.e., they don't know the execution order. They can be generated at run-time or at static-time. In the first case, they contain all methods effectively called. A static call graph instead contains all calls that may be executed. This interface has two methods:

1. **Call resolve(Unresolved call)**: remember when we discuss parsing a function call? During the creation of the LiSA program, we didn't have all the information necessary to find what the called function is exactly. This method of the `CallGraph` permits the exploitation of the type information to find a possible target of an `UnresolvedCall` from the CFGs of the program and returns it.

2. **AnalysisState<H, V> getAbstractResultOf(CFGCall call, AnalysisState<H, V> entryState, Collection<SymbolicExpression>[] parameters)**: evaluates the abstract result of a CFGCall (that is an implementation of CallWithResult) knowing the entry state. The parameters of the call are represented by SymbolicExpressions.

LiSA provides two implementations of the CallGraph which differ in the way of construction: RTACallGraph and CHACallGraph. To better understand why there is more than one way of constructing a Call Graph, just think about object-oriented program analysis. If we desire a precise Call Graph, we need to reason about which function is called at a given program point. But this is not easy to determine due to the polymorphism. Suppose that we are writing this trivial Java program:

```

1 public class Main {
2     public static Number getNumber(String type) {
3         if (type.equals("long")) {
4             return Long.valueOf(5);
5         } else {
6             return Integer.valueOf(5);
7         }
8     }
9
10    public static void main(String[] args) {
11        Number n = Main.getNumber(args[0]);
12        Double d = Double.valueOf(10.0);
13        float f = n.floatValue();
14    }
15 }
```

Code 3.3: Java class example

At line 13 of [3.3](#), we are calling the method `doubleValue()` of an instance of `Number`. The class `Number` has subclasses, like `Integer`, `Long`, `Short`, `Float`, etc. What `floatValue()`s are possibly called at this point? Let's see the two algorithms that LiSA uses to determine them to build the static CallGraph.

CHACallGraph CHA stands for Class Hierarchy Analysis, and it is very simple: when we call a method `m()` on a declared type `T` (i.e., `x.m()` where `x` is an instance of `T`), the CHA algorithm will consider `T` and all the subclasses of `T` that implements `m` as possible target calls, and it will add

them to the Call Graph. In the example before, program [3.3](#), following the mentioned rule, the CHACallGraph will add edges to all the floatValue() method of all subclasses of Number that override it (so we have an edge to AtomicInteger.floatValue(), another to Integer.floatValue(), another one to Byte.floatValue() and so on). This algorithm will generate a correct call graph (it contains edges for all calls the program may execute), but it is very imprecise: most calls in the graph will never be executed.

RTACallGraph An improvement and efficient (but still imprecise) algorithm is the RTA (Rapid Type Analysis), used by the RTACallGraph. It considers not all the subclasses of T but only subclasses that the program instantiates. In this case, the analysis keeps track of the instantiated types. What's the problem here? Take into account (again) the java program [3.3](#). The call graph contains edges only to Integer.floatValue() and Long.floatValue() but also to Double.floatValue() because we have a Double Object (subclass of Number) in the program, even if with a simple look at the code we as human are sure that Double.floatValue() will never be called.

3.5.5 Semantics of Statements

Our Statement class has a method called semantics that permits us to compute its actual meaning. Starting from an AnalysisState and a CallGraph, this method will return a new AnalysisState that contains the current Statement's semantics. This method can rewrite a LiSA program's statement expression into a SymbolicExpression.

3.6 Architectural Scheme

Figure [3.12](#) shows the architecture of LiSA.

Since a program can be written in multiple languages, it is necessary to split the input program P (the program we want to analyze) into programs P^i . Each program is written in a programming language, L^i . The front end L^i (front end able to recognize language i) parses the program P^i , transforming it into a Lisa program P^i_L . Then, every produced Lisa program is merged into one bigger program P_L , that is, the union of all program P^i_L . P_L is the entry point of the LiSA Engine. Other than a LiSA program, the framework can accept in input also a set of configurations that permits tuning the analysis, setting, for example, a custom domain or declaring the desired output

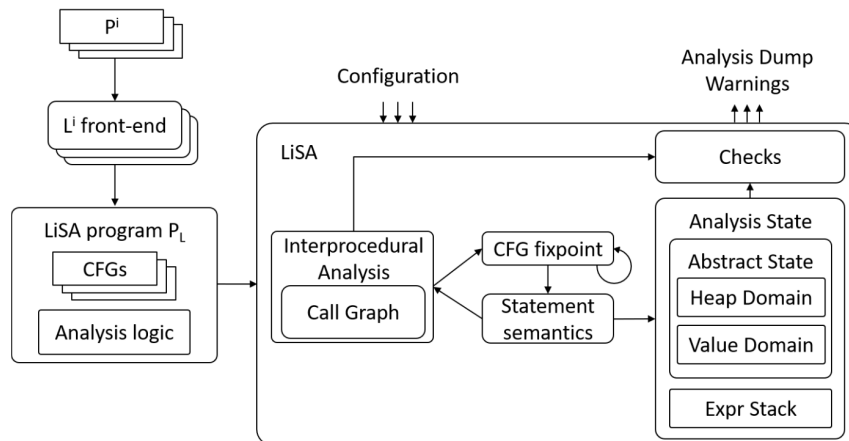


Figure 3.12: Architectural scheme of Lisa. Figure taken from [29]

format.

LiSA then performs an Interprocedural Analysis, computing the program’s fixpoints (to extract the semantics) and the results of function calls with the help of the call graph.

During the analysis, it uses an Abstract State for computing the abstract value. As we said, Abstract State models the memory state of the program and consists of a Heap Domain plus a Value Domain. The Expr Stack is a set of symbolic expressions computed from the evaluation of the last processed statement. The Expr Stack and the Abstract State compose the Analysis State. The last step of the analysis consists of calling the Checks. Checks visits every unit and CFG of the program and produce useful files and warnings. We will talk about them in the next section.

3.7 Checkers

A feature that LiSA provides is the possibility to write checkers. Checkers are tools that permit the definition of some conditions about the syntax or the semantics of the programs, and they can raise some warnings dumped into a JSON file. LiSA defines two types of checkers:

1. **SyntacticCheck**: This is a check that can exploit the program’s syntactic structure.
2. **SemanticCheck**: A check that permits the exploitation of not only the

syntactic structure of the program but also to reason about the semantics. This is done by using the results of the analysis.

3.7.1 How they Works

Let's see the methods provided by the interface `Check`. Note that this interface is parametric to the tool used during the analysis, defined by capital `T`. The tool can log the warnings, and it can be used to obtain additional information, such as the state of the analysis. Generally, a `SyntacticCheck` uses a simple tool with only logging purposes, while a `SemanticCheck` has an extended tool that also contains an analysis state and permits fetching the current fixpoint computation for extracting the semantics to work with. The methods provided by the interfaces are hooks that are called internally by the analysis engine.

1. **`void beforeExecution(T tool)`**: this method is invoked once before the program analysis. The main purpose of it is for initialization and setup of the checker.
2. **`boolean visitUnit(T tool, Unit unit)`**: it will visit an unit of a program. It returns true if the unit needs to be visited: in this case, the globals of the unit are visited. Otherwise, it will not propagate the check in the current unit.
3. **`void visitGlobal(T tool, Unit unit, Global global, boolean instance)`**: this function will visit a global variable of a unit by knowing if the global is an instance variable of the unit or not.
4. **`boolean visit(CheckTool tool, CFG graph, Statement node)`**: it will check a statement of the graph (the current cfg under analysis). Returns true if the visits should continue to subsequent statements of the given CFG.
5. **`boolean visit(CheckTool tool, CFG g)`**: it will check a cfg. It returns true if the visit should continue inside the cfg to analyze its statements. For example, this method can be used if we want to check only special functions with a given name: if the name of the cfg is the name of the function that we want to exploit, returning true, the checker will visit the cfg. Otherwise, returning false, this function is skipped by the checker.

6. **boolean visit(CheckTool tool, CFG graph, Edge edge)**: it will visit an edge of a CFG: returns true if the visits should continue on this graph.
7. **void afterExecution(CheckTool tool)**: this function works like beforeExecution: it is called once when the analysis on the program ends. It is used, for example, to perform some logging aggregation.

3.7.2 An Example

We will now see a simple example of a syntactic check for better comprehension. We want to check if a function called "test" exists inside a Python code. The implementation of this checker is defined in Code [3.4](#).

```

1 public class TestFunctionFinder implements SyntacticCheck
  {
2     private int count;
3     @Override
4     public void beforeExecution(CheckTool tool) {
5         // initialization
6         count = 0;
7     }
8     @Override
9     public void afterExecution(CheckTool tool) {
10        tool.warn("Found " + count + " functions with name
11            test");
12    }
13    @Override
14    public boolean visitUnit(CheckTool tool, Unit unit) {
15        // A unit could be a Class. This method should
16        // return true since a function named test could
17        // be defined inside it.
18        return true;
19    }
20    @Override
21    public void visitGlobal(CheckTool tool, Unit unit,
22        Global global, boolean instance) {
23        // Here, we do nothing: we are not considering
24        // globals (i.e., variables)
25    }
26 }

```



```

21     @Override
22     public boolean visit(CheckTool tool, CFG graph) {
23
24         if (graph.getDescriptor().getName().equals("test")
25             ) {
26             count += 1;
27             tool.warnOn(graph, "test function found!");
28         }
29         // We want to perform analysis only on the
30         // function's signature: it is not necessary to
31         // visit the cfg.
32         return false;
33     }
34     @Override
35     public boolean visit(CheckTool tool, CFG graph,
36         Statement node) {
37         return false;
38     }
39     @Override
40     public boolean visit(CheckTool tool, CFG graph, Edge
41         edge) {
42         return false;
43     }

```

Code 3.4: A SyntacticCheck able to find functions named test

The explanation of the code is straightforward. Feeding the analysis with the next Python code (Code [3.5](#)):

```

1 class x:
2     def test():
3         print("test")
4     def test(x):
5         print("test")
6 def test():
7     print("test")
8 def not_test():
9     print("not_test")

```

Code 3.5: Example of a Python code

It will output this report in .json format (Code [3.6](#)), dumping all the warnings logged with the CheckTool:

```

1 {
2   "warnings" : [ {
3     "message" : "[ 'test/main4.py':2:4] on 'untyped x::test
4       ()': [CFG] test function found!"
5   }, {
6     "message" : "[ 'test/main4.py':4:0] on 'untyped x::test
7       (x* x)': [CFG] test function found!"
8   }, {
9     "message" : "[GENERIC] Found 3 functions with name
10      test"
11   } ]

```

Code 3.6: The report generated by [3.4](#) after analyzing the code in [3.5](#)

3.8 PyLiSA SARL

PyLiSA has a special SARL (Static Analysis Refining Language) [\[29\]](#). It is a domain-specific language that comes to our help when there is a need to define the behavior of framework and/or libraries [\[21\]](#), improving the precision and soundness of the analysis. SARL can be used to define code annotations and, for our use case, to define custom objects and methods of libraries that could be called in a program. The PyLiSA SARL comes with a simple grammar and its front end, and it is used (for now) for modeling standard-library methods and objects, providing, in addition, the declaration of some numpy and pandas methods. This approach permits to include in the analysis of the program external Python libraries in an effortless way improving the precision of the analysis without changing the code and the front end of the library. This SARL is very straightforward, and the specification can be written in a simple text file.

```

1 library numpy:
2   location numpy

```

```
3     method array: it.unive.pylisa.libraries.numpy.  
      Array  
4         libtype numpy.NDArray*  
5         param array_like libtype Object*  
6     sealed class numpy.NDArray:  
7         instance method reshape: it.unive.pylisa.  
          libraries.numpy.Reshape  
8         libtype numpy.NDArray*  
9         param a libtype numpy.NDArray*  
10        param shape libtype Tuple*  
11       instance method reshape: it.unive.pylisa.  
          libraries.numpy.Reshape  
12       libtype numpy.NDArray*  
13       param a libtype numpy.NDArray*  
14       param x type it.unive.lisa.type.common.  
          Int32Type::INSTANCE  
15       param y type it.unive.lisa.type.common.  
          Int32Type::INSTANCE
```

Code 3.7: Snippet of a PyLiSA SARL file entry

Code [3.7](#) shows a snippet of the PyLiSA SARL for specifying what to do with some methods of the numpy library. This tells the analysis, for example, that inside the numpy library (that is parsed in a `ClassType` instance) exists an array method that takes in input a pointer to a generic `Object` ((`array_like`, line 8) and returns a pointer to a `numpy.NDArray` Object (line 7). The semantics of the method (i.e., what the method does) is modeled through Lisa's CFG expression statement and, for this specific method, is defined inside `it.unive.libraries.numpy.array` (line 3). During the analysis, if we encounter a call to the method `numpy.array` with an object as an argument, LiSA will know how to handle it even if we don't have an explicit CFG implementation of the method inside the program's source code. A Class is defined on line 9 of [3.7](#). This class has name `numpy.NDArray` and the semantics of some instance methods are provided.

Although it is almost impossible to define a precise semantics of a method of a library (some methods from external libraries are complex and maybe call other methods from other libraries), one could just provide an abstraction of the results of the called method as a tradeoff.

3.9 Running a LiSA Analysis

Running LiSA is nothing difficult.

```

1 PyFrontend translator = new PyFrontend("/path/to/file/main
  .py", false);
2 Program program = translator.toLiSAProgram();

```

Code 3.8: Get a LiSA program from a source code

Firstly, we need to create a LiSA program from the front end, and the two lines of code in [3.8](#) tell how.

```

3 LiSAConfiguration conf = new LiSAConfiguration();
4
5 conf.optimize = false;
6 conf.workdir = "test-ros-output";
7 conf.serializeResults = true;
8 conf.jsonOutput = true;
9 conf.analysisGraphs = LiSAConfiguration.GraphType.
  HTML_WITH_SUBNODES;
10
11 conf.interproceduralAnalysis = new ContextBasedAnalysis
  <>();
12 conf.callGraph = new RTACallGraph();
13 conf.openCallPolicy = ReturnTopPolicy.INSTANCE;
14
15 conf.syntacticChecks.add(new TestFunctionFinder());
16 conf.semanticChecks.add(new ROSComputationGraphDumper());
17
18 FieldSensitivePointBasedHeap heap = new
  FieldSensitivePointBasedHeap();
19 TypeEnvironment<InferredTypes> type = new TypeEnvironment
  <>(new InferredTypes());
20 ValueEnvironment<ConstantPropagation> domain = new
  ValueEnvironment<>(new ConstantPropagation());
21 conf.abstractState = new SimpleAbstractState<>(heap,
  domain, type);

```

Code 3.9: Create a LiSA Configuration

After that, we must create a LiSAConfiguration (code [3.9](#)). This Object instructs LiSA how to do the analysis, i.e., what domains (lines 18-21) and call

graph constructor algorithm (line 12) it must use, how to handle open calls (line 13), what instance of interprocedural analysis to use (line 11). There is also a flexible control about the output (lines 6-9): do you want a full report in HTML representation? Or do you just need a .json file? In what folder the output files must be saved? A semantic or syntactic checker can be added in a LiSAConfiguration with just a line of code (lines 15-16). We will explain in details this configuration in Chapter 6.

```
1 LiSA lisa = new LiSA(conf);
2 lisa.run(program);
```

Code 3.10: Run LiSA with a configuration

What remains to do (code [3.10](#)) is to instantiate a new LiSA Object with a LiSAConfiguration and run the analysis by calling the method run with our LiSAProgram as a parameter.

Chapter 4

ROS

*"Robot: a machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer."*¹

If you take a dictionary and search for the meaning of the words Robot, probably you will find a definition like the one above.

Robots nowadays are everywhere: in our homes (for example, these little boxes that automatically go around the room and clean the pavement), in our gardens (the mini lawn mower that goes back and forth with the sole principle of keeping the grass cut) and also in our sky (the drone that your nerdy friend bought).

Note that this list is not exhaustive: robots are used in a wide range of fields like medical (for example, to decontaminate a non-human safe area [33]), environmental science (such as underwater drones for investigating underwater cultural sites [5]) and for space exploration (the famous NASA's rovers²).

In this chapter, we will provide a brief introduction to ROS (Robot Operating System) [34] [28]. We will focus on the Python library of ROS2, named rclpy (our companion with PyLiSA in the next chapter), and we will see a simple example of a minimal program. Note that in the ROS ecosystem, there are two different frameworks: ROS1 [34] and ROS2 [28]. This thesis will discuss ROS2 and refer to ROS2 as ROS.

¹<https://canadacommons.ca/topics/robots/>

²<https://spaceplace.nasa.gov/mars-rovers/>

4.1 Introduction to ROS

ROS stands for Robot Operating System. As mentioned in [34] (from which this section takes inspiration) and in the official ROS documentation³, ROS is not a traditional operating system. However, it is a meta-operating system: it is a sort of abstraction of the underlying OSs of machines of a heterogeneous cluster that permits easily to exchange messages to other machines of the cluster, providing a structured communication layer and an additional hardware abstraction. It is a set of software libraries and tools for building robot applications and networks. This framework can help resolve specific issues regarding software development for robots. We will now provide a detailed list of some of them.

- **Distributed Computation** Robots can communicate with each other in a distributed way, cooperating to resolve a common and shared task. Additionally, it is often the case that robots internally are composed of several machines that control a specific subset of robot sensors. A communication mechanism is needed, and the ROS framework defines a P2P infrastructure among robots: computational logic relies on entities called nodes. These nodes communicate by exchanging messages and can run across different computers.
- **Reusage of Software** Some standard algorithms that solve tasks like motion planning (i.e., move objects from a source to a destination by finding a valid sequence of configurations) or mapping (constructs a map of the environment where the robots are) are already implemented in ROS. Since the communication interface that ROS provides is gaining attention in academia and industry, many people actively maintain and update ROS to guarantee compatibility of the framework with the latest hardware and operating systems. Also, additional libraries (plugins) that extend the core framework are easy to find. This permits ROS developers to write their code with the help of already-defined packages and algorithms.
- **Multi language** The client framework library of ROS is called rcl and is written in C language. To support other programming languages, some binding libraries are available. The main peculiar one for our purpose is the rclpy library, the Python binding for rcl, but it also exists bindings for Java (rcljava), C++ (rclcpp), and others. This is one of

³<https://docs.ros.org/en/humble/index.html>

the key differences between ROS1, which has the client code written independently in each language. These binding libraries are called ROS Client Library.

- **Open-Source** ROS is free to use, and the source code is publicly available under the term of a BSD license.
- **Security** ROS has an internal security system that permits the usage of encryption and authentication mechanism. Some Access Control Policies manage node communication [10] that defines which nodes can communicate about what. Note that these security features are not enabled by default.
- **DDS as a middleware** DDS (Data Distribution Service) is an implementation detail of ROS 2. This means that underlying the client ROS API there is an Abstract DDS API that communicates with a DDS instance. ROS2 has support for different DDS vendors. The main idea to use it as a middleware is that the DDS can be substituted easily without code changes. What is a DDS? We will talk about it in a moment.
- **Host OSs** ROS can be hosted in Linux, macOS, Windows, or RTOS machines.

4.1.1 Brief Definition of DDS

DDS is a standard messaging protocol that provides a publish-subscribe transport. Nowadays, it is adopted by various industries: for instance, autonomous vehicles, space systems, air traffic management, and Robotics. DDS uses IDL (Interface Description Language) for message definition and serialization. IDL enables the description of message interfaces in a manner that is not tied to a specific programming language. This peculiarity permits DDS instances to work as a bridge between two (or more) languages, enabling communication between software components written in more than one language. DDS is mainly used for real-time data-centric pub/sub communication in distributed systems. DDS relies on RPC (Remote Procedure Call)⁴ framework that provides a request/response style transport. DDS is used in ROS because of its reliability and flexibility. It supports UDP and TCP connections and provides a Quality Of Service control mechanism for tuning the communication based on network characteristics.

⁴<https://www.omg.org/spec/DDS-RPC/>

4.2 Concepts and Terminology

In this section, we will briefly see some concepts regarding the ROS ecosystem and set the terminology we will use in the following sections and chapters.

4.2.1 The ROS Domain

ROS Domain is a subnetwork of entities (e.g., robots). Entities can communicate and can be seen only by entities sharing the same ROS Domain. An integer ID defines a ROS Domain.

4.2.2 The ROS Graph

The core of the ROS system is the ROS Computation Graph. It is simply a network of entities, called nodes, that can communicate with each other by message exchange. The computational graph tells us the nodes that form the P2P network and how they interact with each other (i.e., which nodes communicate with).

4.2.3 Nodes

Nodes are the main entities (i.e., the vertices) of the ROS Graph. They are single-purpose processing modules within ROS. To communicate with other Nodes, a Node uses the ROS Client Library classes and methods to interact with the underlying middleware and send messages. A Node can use or define Actions and Services. Generically a Node has a name that identifies it, plus some additional configurable parameters that permit tuning its behavior. The behavior of a Node is reactive: it is triggered by events. An event could be internal to the system (i.e., receiving a message or an alarm ring) or external (a spontaneous impulse). If Nodes are vertices of the ROS Graph, we need to define what edges are (spoiler: we could have different types of edges!).

4.2.4 Topics

The primary type of edges ROS provides to connect nodes is Topic. Topics permit communication between two Nodes. They work in a flexible pub/sub structure. A Node can subscribe to (or listen to) one or more Topics and

publish on (or talk on) one or more. We call Publisher a Node that publishes something in a Topic and a Subscriber a Node that is listening in a Topic. Topics support many-to-many relationships: in a Topic, we could have zero or more than zero Publisher and zero or more than zero Subscriber. The unit of information pushed and pulled in a Topic is called message. Topics are used for continuous data transfer between Nodes.

4.2.5 Messages

Topics permit the exchange of messages between Nodes. ROS has an internal Message structure that defines how data is exchanged between Node using Topics. We can communicate the most basic types, for example, String, Integer, Boolean, and so on, or a combination of them to form a custom and user-defined Message structure.

4.2.6 Parameters

Parameters are name-value pairs, a configuration item of a Node. Parameters value could only be a Bool, a String, an Integer, a Double, an array of them, or an array of Byte. Parameters define the internal attributes of Node.

4.2.7 Services

Instead of using Topics, Nodes can communicate with each other using Services. They are used when we want synchronicity and with a guaranteed response from a Node using a Request/Reply mechanism. When using Services, we don't perform preemption: Services block calls. Services can be used, for example, for setting the parameters of a Node remotely or to perform quick calls. Services use the YAML format for Request and Response bodies.

4.2.8 Actions

Another way of communication is through Actions. Actions and Services are very similar in definition, but the main difference is that Actions are used when we want something asynchronously, implementing a preemptive goal-driven behavior. Actions are used when we have a long task to execute. Actions are a sort of Client/Server model, and they work by setting a goal for the server from a client. The server then provides regular feedback when it

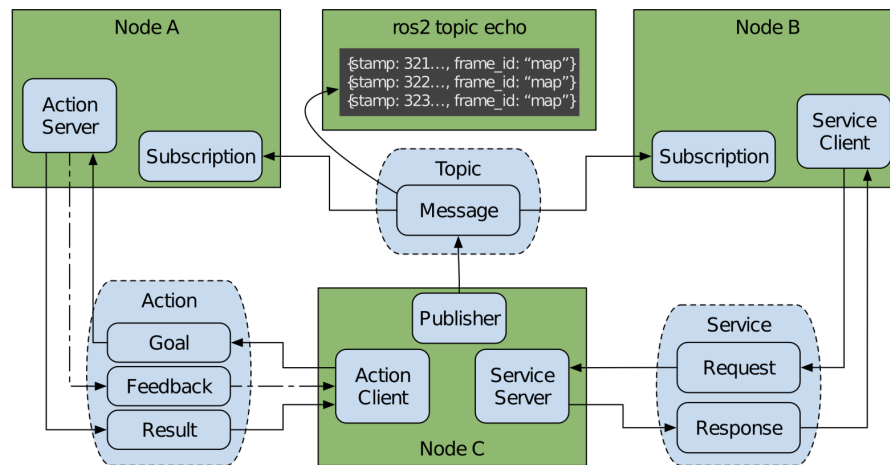


Figure 4.1: An overview of the ROS interfaces

reaches checkpoints while executing the Action. Like Services, Actions use the YAML format to define bodies of Goals, feedback, and final response.

4.2.9 Discovery Process

The underlying DDS takes care of the discovery of new Nodes automatically. Using DDS, a Node:

- When it is started, it advertises its presence to other network Nodes with the same ROS Domain ID. The Nodes that receive the message will respond with information about themselves.
- Periodically, it advertises its presence to perform connection with new-found entities.
- When it goes offline, it advertises other Nodes of its disconnection from the network.

Figure 4.1 is taken from [28] and can help us better understand how ROS ecosystem entities interact. The figure depicts three Nodes (Node A, Node B, and Node C) that interact with the usage of Topic, Action, and Service. We can deduce that a Node could have an internal Action Server or Client, a Subscription or a Publisher, and a Service Server or Client. He could also

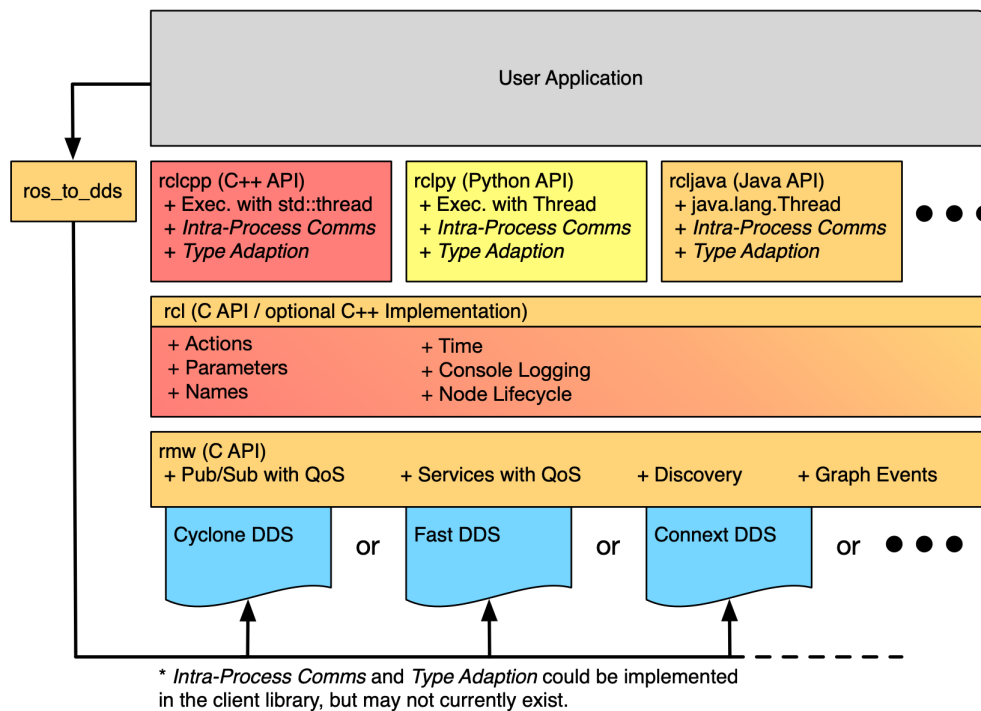


Figure 4.2: ROS Internal API Architecture Overview

have a combination of them and more than one of these entities. For example, we can see that Node C has a Publisher that sends messages on a Topic T. Node A and Node B has Subscribers that fetch messages from T. In this way, Node C can send messages to the other Nodes by pushing them to T. Node A has also an Action Server that Node C uses with his Action Client. Firstly, Node C will set the Goal of the Action (simply Node C tells the server what it should achieve and how it should do it). Node A will analyze the Goal and start achieving it, sending feedback to Node C during the process. When he reaches the Goal, it sends the result to the client. Node C also has the capability of being a Service Server. A Service works like a web service: it processes requests from Clients (in this case, there is only one Client, Node B) and sends back the response.

4.3 ROS API Architecture

Figure [4.2](#) shows the internal architecture of the ROS ecosystem. Following a bottom-up analysis, we find the rmw API: the interface that permits com-

munication with the underlying DDS middleware (in cyan). The DDS middleware is responsible for all the things regarding communication and discovery; that is to say, it contains the implementation of the publisher/subscriber mechanism, the service request-reply mechanism, and the serialization of messages. This layer can be seen as an abstraction of the communication middleware.

The next layer is the aforementioned rcl library. rcl does not communicate directly with the underlying middleware but is built on top of rmw. The purpose of this layer is to give a common implementation for complex ROS concepts (for example, nodes and the ROS Computational Graph) and algorithms that client libraries may use. Client libraries (the layer just above rcl) binds to this C library and provides a simple-to-use extendable library to the developer. For example, a developer that wants to write an application using the ROS framework (the higher layer in the figure) in Python can import the rclpy library.

A precise reader can observe that, on the figure's left, an orange rectangle labeled `ros_to_dds` connects the User Application directly with the underlying middleware. This models the possibility of using specific external packages that permits direct communication with the DDS for an application that requires it.

4.4 About DDS-Security and SROS2

We want now to discuss the security of DDS. DDS is just a specification, but this specification doesn't say too much about security⁵. By its design, the standard DDS specification is not concerned with security. To overcome this, another specification aims to add security support to the DDS one. This specification is called DDS-Security⁶ and defines some extensions to the DDS that improve security. In particular, it provides five plugins that, when combined, give Information Assurance to the DDS system. Information Assurance is a set of measures that protect information systems by ensuring availability, integrity, authentication, confidentiality, and nonrepudiation [40]. These plugins are:

1. **Authentication Service Plugin:** it permits the addition of a mutual authentication mechanism establishing a shared secret to verify the identity of an application or a user operating on DDS.

⁵<https://ubuntu.com/blog/security-vulnerabilities-on-the-data-distribution-service-dds>

⁶<https://www.omg.org/spec/DDS-SECURITY/1.1/>

2. **AccessControl Service Plugin:** adds the capability to define rules to enforce policy decisions regarding what an authenticated user can do. It can be used, for instance, to define what domains a user can join or which Topics it can publish or subscribe to. For each domain participant (i.e., for each ROS Node), it requires two signed XML files:
 - **Governance.xml:** it specifies how the domain should be secure.
 - **Permissions.xml:** it specifies the permissions of the domain participant.
3. **Cryptographic Service Plugin:** implements cryptographic operations such as encryption, decryption, hashing, and digital signatures.
4. **Logging Service Plugin:** supports inspection and auditing of all DDS security events the system could generate. This service provides the capability to log all events regarding security, including violations and security errors.
5. **Data Tagging Service Plugin:** permits to tag data samples. Tagging has several uses; for example, it can be used for access control (by granting access based on the tag) or for prioritizing messages, or it can be used directly by the application.

By default, none of those mentioned above DDS security features of DDS are enabled in ROS2⁷. It is very trivial to enable them (it requires changing a flag in the environment variable), but setting up all these features correctly in a ROS ecosystem is not an easy task, and its complexity is prone to human errors. To make things easier, the SROS2 toolset was proposed [42]. This toolset comes with a security methodology for robotics applications that facilitates a secure DevOps model in this field.

Among all the features SROS2 provides, we want to mention the ROS2 APIs extension, which permits introspecting the computational graph at the networking level. This simplifies the threat modeling of a ROS application, providing information about all the entities that live inside the ROS computational graph and monitoring their interaction.

Furthermore, SROS2 CLI features a tool to help developers set up DDS-Security plugins. For instance, it creates:

- Certificate Authority for Authentication

⁷http://design.ros2.org/articles/ros2_dds_security.html

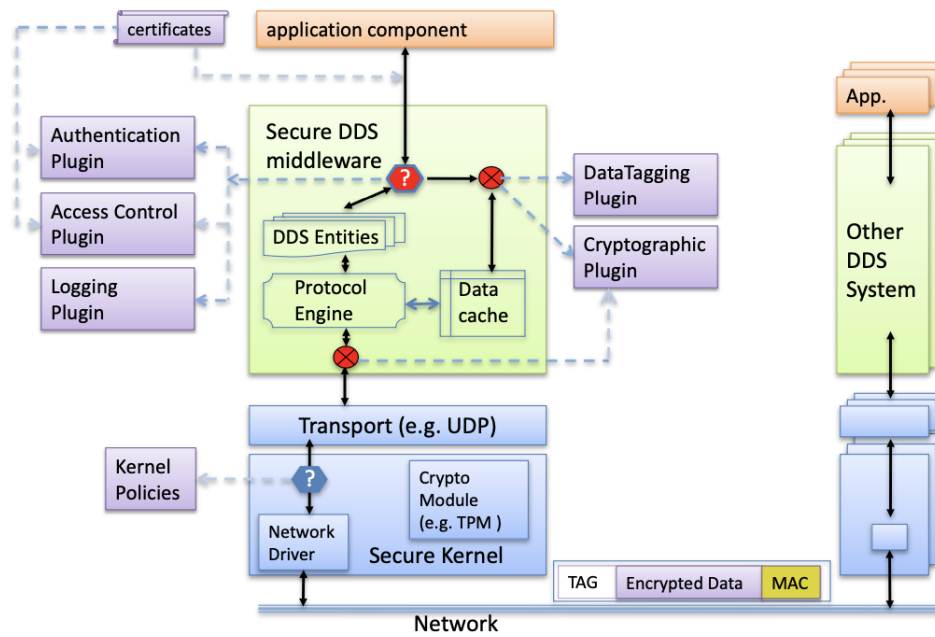


Figure 4.3: Overall Architecture for DDS Security (figure taken from the official DDS Security specification)^a

^a<https://www.omg.org/spec/DDS-SECURITY/1.1/>

- A governance file that will encrypt all DDS traffic by default
- Improved Access Control Policy files

4.5 rclpy

The main component of this thesis is the rclpy library: as we already mentioned, it is a Python library that binds the rcl library. We will now see how it works, pausing on the most important methods and classes that are crucial for the next chapter and omitting all the rest: it is not the purpose of this thesis to provide documentation of the rclpy library. The official one⁸ is well written from the writer's point of view, and if more is needed for the reader, there is also the official wiki with tutorials, definitions, and some examples.

4.5.1 Application Life Cycle

A ROS Program Life Cycle consists of these steps:

1. **Initialization** In this phase, a Context is initialized. The Context contains all the information about the status of the application, and it is used internally by the application's entities to understand what's going on at an exact moment in the app. The initialization must be done before any Node creation.
2. **Node(s) creation** The next phase is the Node(s) creation, in which we create one or more Nodes that can be used to create Publisher or Subscriber. Nodes have an associated Context.
3. **Process Nodes Callbacks** Nodes have callbacks, which permit the definition of what must be done when something happens, like, for example, the ring of a Timer or other events. For example, when a Node registers a Subscriber in a Topic, it must specify what logic must be performed when a Message arrives. When we are ready to activate the callbacks, we need to tell the associated Context that Nodes can be called the callbacks, and this can be done in different ways: for example, callbacks can be processed in a block way (i.e., the main application will be listening on triggers until the Context is shut off), or it can listening and execute only the first callback that found. In the latter example, a timeout can be set: if it expires, it stops listening.

⁸<https://docs.ros2.org/foxy/api/rclpy/>

4. **Shutdown** When we are done with using Nodes, the Shutdown phase must be started. In this phase, we close the Context, and all Nodes of the latter will be invalidated.

4.5.2 The rclpy Module

The rclpy module is the main module of the library. Here we will see what it has to offer. Please note that for brevity and concision, optional function parameters are omitted from the signature, and the most important ones are detailed in the description of the function.

1. **rclpy.init(...)**: it will initialize the Context passed as a parameter (if no Context is present, it will work on the default one).
2. **rclpy.shutdown(...)**: it will shut down the Context passed as a parameter (if no Context is present, it will work on the default one like rclpy.init).
3. **rclpy.create_node(node_name, ...)**: it creates and returns an rclpy.node.Node Object with the given name. Some other optional parameters can be passed, for example, the Context that the new Node must belong to or the namespace. A namespace is a string prefix that will be prepended to the Topic name when it declares Subscribers and Publishers.
4. **rclpy.spin(node, ...)**: it will execute works on a Node (like callbacks), and it blocks until the Context is shut down.
5. **rclpy.spin_once(node, ...)**: it will execute one item of work on a Node. A timeout can be set, and the function does nothing if, during the period, no work can be found in the Node.

These are the main methods provided by the rclpy main package⁹. The sub-modules provide the definition of ROS Objects like Node¹⁰, Publisher, Subscription and Topics¹¹.

rclpy.node.Node The class Node of module rclpy.node represents a Node in the ROS ecosystem. The core methods that the Node class has are:

⁹<https://docs.ros2.org/foxy/api/rclpy/api.html>

¹⁰<https://docs.ros2.org/foxy/api/rclpy/api/node.html>

¹¹<https://docs.ros2.org/foxy/api/rclpy/api/topics.html>

1. **`__init__(self, node_name, ...)`**: instantiate the Node with the given `node_name`. The optional parameters are the same as `rclpy.create_node` function.
2. **`create_publisher(self, msg_type, topic, qos_profile, ...)`**: create and return a Publisher able to publish on topic Message of `msg_type`, with a `qos_profile` that defines what quality of service policy must the Publisher use.
3. **`create_subscription(self, msg_type, topic, callback, ...)`**: create and return a Subscription able to listen on topic Message of `msg_type`, with a `qos_profile` that defines what quality of service policy must the Subscription use and a callback that is called when the Subscription receives a message.
4. **`create_service(self, srv_type, srv_name, callback, ...)`**: create and return a Service server with the given `srv_name`. The callback is called when the Service receives a request from a client
5. **`create_client(self, srv_type, srv_name, qos_profile, ...)`**: create and return a Client able to communicate with a Service server with name `srv_name`.
6. **`create_timer(timer_period_sec, callback)`**: create and return a Timer that every `timer_period_sec` seconds will call the callback function.
7. **`declare_parameter(self, name, ...)`**: declare a Parameter with the given name in the Node. Optionally, the parameter value can be used to set a value of the Parameter.
8. **`declare_parameters(self, namespace, parameters, ...)`**: declare in a Node a set of Parameters with the name taken from the Parameters argument array, prefixed with namespace following a simple expansion rule (`namespace.name`). If the namespace is the empty string, the dot is not prefixed to the parameter name.
9. **`undeclare_parameter(self, name)`** remove the Parameter with the given name from a Node.
10. **`add_on_set_parameter_callback(self, callback)`**: register a callback triggered when we set a parameter in a Node. Note that the `undeclare_parameter` function will not cause the callback.

11. **remove_on_set_parameter_callback(self, callback)**: remove the given callback from the list of callbacks triggered when we set a parameter.
12. **remove_on_set_parameter_callback(self, callback)**: remove the given callback from the list of callbacks triggered when we set a parameter.
13. **get_name(self)**: returns the name of the Node.
14. **get_node_names(self)**: returns a list containing the names of discovered Nodes.

This list is not exhaustive: there exists, for example, methods for destroying a client, a publisher, a subscription, a timer, and the Node itself, and also methods that permit to find services, topics, publishers, and subscribers of discovered nodes.

rclpy.publisher.Publisher Since the method `create_publisher` of Node should create Publisher, we don't list the `__init__` method here.

- **get_subscription_count()** returns the number of subscribers that this Publisher has.
- **publish(self, msg)** publish the message in the Publisher's Topic. It will raise an error if the msg is not of the type provided during the construction of the Publisher.
- **destroy(self)** destroy this Publisher.

rclpy.publisher.Subscription Subscriptions like Publisher should be created from a Node using the appropriate method. This class does not have relevant methods except `destroy()`, which works like the Publisher's counterpart.

rclpy.service.Service This class is a wrapper for the underlying rcl Service.

- **send_response(self, response, header)**: Sends a Service responses.
- **destroy(self)**: destroy this Service.

rclpy.service.Client This class represents a Service Client.

- **call(self, request)**: make a request to a Service Server, and it will wait until the Service response. This is a blocking call, and the documentation says that calling this method in a callback could lead to a deadlock.
- **call_async(self, request)**: call a Service and return a Future that completes only when the request completes.
- **remove_pending_future(self, future)**: remove a Future that is waiting the Service response.
- **service_is_ready(self)**: returns True if the Service is ready to accept requests.
- **wait_for_service(self, ...)**: waits until the Service becomes ready.
- **destroy(self)**: destroy this Service Client.

rclpy.action Other important classes are ActionClient and ActionServer, which relies on the rclpy.action module. We let the official documentation ¹²¹³ the honor of describing the implementation details. Actions and Services are not important parts of this thesis since we will focus more on the Publisher/Subscriber communication interface. However, we hope that the reader has understood how the library works in general.

4.6 How to Use rclpy: an Example

After this overview of the ROS ecosystem and the rcl library, it is time to discuss a simple but famous rclpy example from the official documentation. We will see how to declare Nodes and enable communication between two Nodes using the Publisher/Subscriber Message exchange. In this case, we will have two programs: one for the Publisher and one for the Subscriber. They are different entities that could stay in different hosts.

Note that we are not going to launch the program. This thesis wants to be something other than a tutorial on how to use ROS. An interested reader is gently redirected to the official tutorial that will describe how to launch

¹²<https://docs.ros2.org/foxy/api/rclpy/api/services.html>

¹³<https://docs.ros2.org/foxy/api/rclpy/api/actions.html>

these source codes and obtain results in detail. Since this thesis talks about static analysis, we want to focus more on the program's semantics, which can be achieved by looking at the source code with the documentation as a reference. Things that happen run-time are out of scope. After this premise, let's start by analyzing the source code of the Publisher:

```
1 import rclpy
2 from rclpy.node import Node
3 from std_msgs.msg import String
4
5 class MinimalPublisher(Node):
6
7     def __init__(self):
8         super().__init__('minimal_publisher')
9         self.publisher = self.create_publisher(String, '
10             topic', 10)
11         timer_period = 0.5 # seconds
12         self.timer = self.create_timer(timer_period, self.
13             timer_callback)
14         self.i = 0
15
16     def timer_callback(self):
17         msg = String()
18         msg.data = 'Hello World: %d' % self.i
19         self.publisher.publish(msg)
20         self.get_logger().info('Publishing: "%s"' % msg.
21             data)
22         self.i += 1
23
24 def main(args=None):
25     rclpy.init(args=args)
26     minimal_publisher = MinimalPublisher()
27     rclpy.spin(minimal_publisher)
28     minimal_publisher.destroy_node()
29     rclpy.shutdown()
30
31 if __name__ == '__main__':
32     main()
```

Code 4.1: MinimalPublisher example

Code [4.1](#) shows how we can define a Node that acts as a Publisher. First, starting from the main function, we need to initialize the Context calling `rclpy.init(...)` to follow the Application Life-Cycle previously mentioned. Then we instantiate the `MinimalPublisher` class that extends the `rclpy.node.Node` class. During the instantiation, inside the constructor, a call to the Node constructor is made, passing the name of the Node (in this example, "minimal_publisher", line 8). The `MinimalPublisher` constructor allocates a Publisher with the `create_publisher` method of Node (line 9). This Publisher can send Messages of String type, which will publish on Topic with the name "topic". A reference to the Publisher object is stored in a variable of `MinimalPublisher` (`self.publisher`) for easily retrieving it when needed. On line 11, it creates a timer, with the reference of it stored in `self.timer`. This timer will call the callback (function `self.timer_callback`) every `timer_period` (0.5 seconds, defined on the line before). A reference of the `MinimalPublisher` object is stored in variable `minimal_publisher` local to the main function. On line 25, we call the `rclpy.spin` method that, as we said before, will execute the callbacks registered to the Node passed as an argument (in this case, our `minimal_publisher`).

So every 0.5 seconds, the internal Node timer will call the callback, the `rclpy.spin()` method captures it and executes the function. The `timer_callback` function, defined in line 14, creates a String Message and fills the message's field `data` with a string (lines 15 and 16). After that, the message is published on the Topic defined inside the constructor of the `MinimalPublisher` using the `publish` method. Then it will perform some logs and increment a variable that easily captures the meaning of it (it is used as an incremental ID to identify the message inside the string pushed to the Topic).

In the end, we will destroy the Node and call `rclpy.shutdown()` static method for terminating the underlying Context. And this is how a Publisher works in the `rclpy` library. Let's see the Subscriber now:

```
1 import rclpy
2 from rclpy.node import Node
3 from std_msgs.msg import String
4
5 class MinimalSubscriber(Node):
6
7     def __init__(self):
```

```
8         super().__init__('minimal_subscriber')
9         self.subscription = self.create_subscription(
10             String,
11             'topic',
12             self.listener_callback,
13             10)
14
15     def listener_callback(self, msg):
16         self.get_logger().info('I heard: "%s"' % msg.data)
17
18
19 def main(args=None):
20     rclpy.init(args=args)
21     minimal_subscriber = MinimalSubscriber()
22     rclpy.spin(minimal_subscriber)
23     minimal_subscriber.destroy_node()
24     rclpy.shutdown()
25
26
27 if __name__ == '__main__':
28     main()
```

Code 4.2: MinimalSubscriber example

The MinimalSubscriber above (Code [4.2](#)) creates a Node called "minimal_subscriber" on line 21 where inside the constructor allocates a Subscription object capable of listening on Topic "topic" (the same of the MinimalPublisher) Messages of String Type, with a qos_profile equal to 10. What means this? We didn't talk about Quality of Service Profiles, but an integer here means that the Subscriber internally maintains a queue of messages of capacity the integer passed as a parameter. If the inter-arrival time of messages in the queue is less than the execution time of the callback function, then a queue of Messages will be formed. When the queue reaches the capacity (in this case, 10), then the queue is full and new messages are thrown away. The callback function of this Subscriber is the listener_callback function (line 15) that logs the received messages.

Figure [4.4](#) displays the ROS Computational Graph of the minimal example, where rectangular shapes denote Topics, and ovals represent Nodes. Each entity is clearly labeled with its name. With the aid of ROS command-line tools, one can extract the graph at run-time and easily visualize the entities

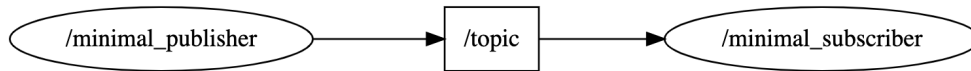


Figure 4.4: ROS Computational Graph of the minimal example

involved in the system and how they communicate. Looking at the figure, we can deduce that the system comprises two Nodes that exchange messages via a single Topic.

Chapter 5

Static Analysis for ROS (An Introduction)

Before this thesis’s leading chapter, we should spend some words about why this thesis exists. This little chapter is divided into two parts: in the first one, we will informally talk about the state of the art of Static Analysis for robotics and, more specifically, for the ROS ecosystem, while in the second one, we will answer the crucial question.

5.1 State of the Art of Static Analysis for Robotics

Static Analysis for Robotics is a challenging task. In the 2013 conference paper [12], the authors argue that there was a lack of static analysis tools concerning robotics applications at the time. In this article, they describe the main static analysis techniques that can be applied to robotics (data-flow analysis, control-flow analysis, model checking, and abstract interpretation), and they observe a lack of standardization in the programming methods of robotics. At this article’s publication date, ROS was a five-year rookie. A decade later, ROS is becoming a defacto standard [2]¹ and something started moving on this topic.

Take, for example, HAROS (High-Assurance ROS Framework) [36], an open-source framework that permits the extraction and the analysis of the ROS Computation Graph at static-time using a metamodel approach. As stated in the official Git Hub repository, HAROS can analyze ROS1 code, and at the time of writing, it does not support ROS2 at all, but the author states that he

¹<https://www.abiresearch.com/blogs/2019/06/10/ros-rise/>

is working on it.

Alhanahnah [4] performed an empirical study regarding Software Quality Assessment of ROS2 Java Project using PMD² and provides a list of related works concluding and observing that only one [27] against the 13 works presented targets ROS2. This work analyzes the DDS security protocol using static analysis tools for finding vulnerabilities and errors in the ROS 2 Ardent Apalone source code. They also discuss ROS2 security features, evaluating them against metrics like throughput and latency to determine if they impact the overall performance.

There are many things to explore in this field, especially for the younger brother of the ROS family.

5.2 Why this Thesis Exists

In this thesis, we want to perform a static analysis of Python code using the rclpy library to extract meanings from ROS source code. The focus is mainly on finding the declaration of Nodes, Topics, Subscribers, and Publishers using the LiSA framework to exploit the ROS Computational Graph at static-time.

Working on this brings with it the following side effects:

1. **Development of a static analysis tool for ROS2 python source code based on Abstract Interpretation.** In the previous sub-chapter, we talked about the state of the art of static analysis for robotics, and we saw that there are no ready-to-use static analysis tools that permit deep analysis of ROS2 Python applications. This thesis could lay the foundations for a new analysis of rclpy projects using LiSA and abstract interpretation.
2. **Improvement of PyLisa front end.** PyLisa is in its first beta version and does not support all Python features. Working on this project led to the implementation of the necessary features (such as, for example, the capabilities to handle Python objects and class extensions and the support of string constant propagation) that enhance the front end.

²<https://github.com/pmd/pmd>

3. **LiSA testing.** The development of this thesis brought the opportunity to perform testing of the core LiSA framework, and this was useful for finding trivial bugs.

Chapter 6

Lisa and ROS: Static Analysis for Robotics

Now we will see how we can analyze a rclpy program using LiSA.

Since rclpy is the ROS2 client library for Python, we are going to use the Python front end for LiSA.

However, PyLiSA is still in its early beta, and some Python features must be officially supported. Unfortunately, unsupported features like the creation of objects and the semantics of Python's `super()` method are heavily used in rclpy programs. This chapter has a dedicated section that addresses that, explaining the necessary front end changes to perform a successful analysis. We divide this chapter into three main sections: firstly, we want to discuss what we will do in detail. Secondly, we will see the main changes required in the PyLiSA front end, and lastly, the main work is presented. Future works are discussed in the next chapter.

6.1 Introduction

A ROS2 application can be specified as a computation graph representing the peer-to-peer network of nodes. The figure presented in [4.4](#) shows an example of this graph, while figure [6.1](#) shows a more complex one. In these graphs, a node is a process, and edges denote a message-passing interaction between these processes.

In this thesis, we provide a way to automatically extract the computation graph of the ROS2 application at static-time just by looking at the semantics of the source code.

This is not a trivial task, as it requires a deep study of the rclpy library and

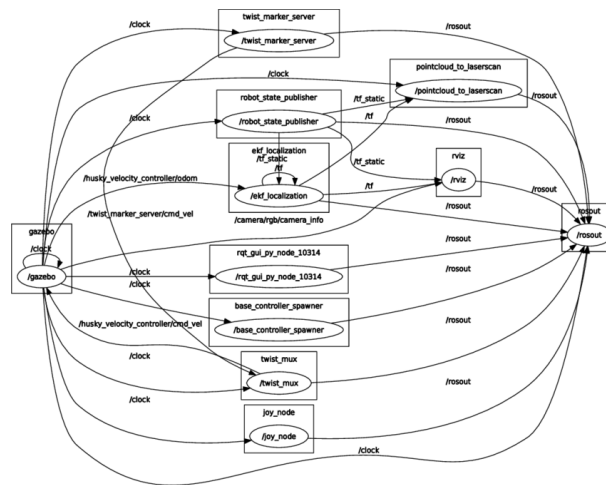


Figure 6.1: An example of a ROS computation graph, taken from [9]

finding a way to implement the semantics of the library classes and methods in LiSA's style. To achieve this, we take advantage of the PyLiSA SARL, allowing us to define the semantics of rclpy elegantly. Using SARL, the Python parser is agnostic of rclpy methods and classes: the front end must parse the SARL file and allocate the classes and methods defined inside it. All rclpy method calls written in the source code are threatened as UnresolvedCall, and the CallGraph will resolve it transparently without tricky changes. The analysis proposed here is the first step to producing more meaningful results: opening the doors at rclpy semantics inside LiSA permits this work to be a starting point for subsequent analysis, as discussed in the next chapters.

At this point, a ROS expert could say: “*But the ROS computation graph can be seen by using the ROS2 command line interface or by using RQT¹!*”. This is true. But note that this is done by analyzing the computational graph at runtime, and we want to achieve this just by remaining static. We already discussed the motivation of this in the previous chapter, and in addition, we would like to say that having the computational graph at static-time permits the validation of the communications concerning the policies defined in the DDS-Security.

The source code of the work can be found here².

¹<https://docs.ros.org/en/foxy/Concepts/About-RQt.html>

²<https://github.com/giacomozanatta/pylisa/tree/feature/gzanatta/thesis>

6.2 PyLiSA Front End Extensions

In this section, we will talk about the main changes that the front end requires.

6.2.1 Object Declaration

During the project's development, there was a lack of meaning in the concept of Object. Since rclpy library relies on objects, it is necessary to provide a way to handle their creation.

Python Classes permit the define specific Object structures. Object constructors rely on special class methods called `__init__`. Code [6.1](#) shows how to declare a class and a constructor in Python. The tricky part is that the constructor and, in general, classes method has a special parameter in its signature that always stays in the first position of the argument list. The first argument represents the Object itself, which is provided implicitly by the call.

Interestingly, the constructor is not called directly when creating an Object, as shown in line 5. For this reason, we need to find a way to tell the front end that we are calling the `__init__` method.

```
1 class Dog:
2     def __init__(self, name):
3         this.name = name
4
5 dog = Dog("Kokoro")
```

Code 6.1: A simple Python Class and an Object instantiation

To achieve this, we create a new NaryExpression called PyNewObj in which we model the semantics of the Object creation.

To allow a smooth integration, the PyNewObj class extends LiSA's NaryExpression. The constructor of the PyNewObj wants a CFG (the control-flow graph where the Object is being constructed), the SourceCodeLocation of this operation, a type (the type of Object being created), and a list of parameters.

The abstract method `expressionSemantics` models the effective creation of objects. Inside this method, we create a MemoryAllocation and a corresponding HeapReference, and we craft (and resolve) a new UnresolvedCall to the actual constructor method. This UnresolvedCall is created by

prepending the parameters of the expressions related to the Object itself. The resolution of this call is responsible for computing its semantics and modeling what's going on inside the constructor's code. In the end, an HeapReference to the Object is inserted into the AnalysisState.

When the front end encounters a method call, it checks if the method's name is a ClassUnit. If it is true, then it creates a PyNewObj. Otherwise, it creates a standard UnresolvedCall.

6.2.2 Object Inheritance Support

Supporting Object inheritance is more complicated.

The first step is to set the ancestors to a ClassUnit: when visiting a class definition, the front end must reason about the superclasses by fetching their names from the class definitions. Then, for every superclass of the current class, it must check if a CompilationUnit exists inside the program being parsed with a name equal to the superclass. If true, it adds the CompilationUnit as an ancestor of the current ClassUnit.

However, more is needed: talking about rclpy, the Node class comes from the external library. In a class declaration like in code [6.2](#), the superclass (in our case, Node) could be an alias (rclpy.node.Node, imported in line 2): so a real class with that name does not exist. This is necessary to give awareness to the front end of all the possible alias, and this is done naively by a map where the key is the alias itself (Node), and the value is the name of the actual class (rclpy.node.Node). When encountering a class declaration with a superclass named S, firstly, we need to check if an entry exists in the map with key S: if it is true, then we substitute S with the value of the map entry, and we use this new value as the superclass name. Otherwise, if doesn't exist the key S in the map, we assume that S is the actual name of the class. However, this approach doesn't cover all the cases, like the one in which we import a library or a module directly inside the current namespace (for instance, from math import *).

```
1 import rclpy
2 from rclpy.node import Node
3
4 class MinimalSubscriber(Node):
5     def __init__(self):
```

```
6 super().__init__("minimal_subscriber")
```

Code 6.2: Python inheritance

When dealing with Object Inheritance, sometimes it is required to call directly a method of a superclass of the Object. In Python, this is achieved using the super special method, as the documentation says. This method can be written in two ways:

- **super(type, object_or_type)**: it provides a proxy object that delegates method calls to a sibling or parent class of the Object³. This permits to support of multiple cooperative inheritances in a dynamic execution environment, enabling the possibility of having “diamond diagrams” where multiple base classes implement the same method.
- **super()**: this can be seen as syntactic sugar of the other super method. In this case, the proxy object returned resolves the calling of methods to the immediate parent of the current class. Line 6 of [6.2](#), produces the same output of “super(Node, self)”.

The main idea of handling these special methods is to expand the second one to have only one class in which we define super semantics. Furthermore, this approach permits circumnavigating the difficulty of modeling the semantics of this call since, without parameters (like an object to work on), it could be infeasible to handle.

The super method is declared inside the SARL file. It is a BinaryExpression, and inside the semantic method, we perform a conversion of the type of the second parameter (the Object) to the type passed as the first parameter. During the parsing, the front end must capture the simple super (the super without parameters). This is because it requires additional information (i.e., the superclass and the Object itself) to correctly convert the call to the expanded super.

This is done by introducing a Java Object (SimpleSuperUnresolvedCall) that extends UnresolvedCall. This works likely the standard UnresolvedCall, except that if it can't resolve super(), it will build and resolve the expanded version. The idea is to first resolve the simple super method call by studying the Python behavior: if a method with the name super without arguments exists, then Python will not call its internal method but the one defined by

³<https://docs.python.org/3/library/functions.html#object>

the user.

When the front end finds a method called `super` that has no arguments, then it creates the `SimpleSuperUnresolvedCall` instead of the standard `UnresolvedCall`.

6.2.3 String Constant Propagation

Some changes were made to the `ConstantPropagation`. Since the name of a node (or of a topic) comes from a previously defined variable (or by a method argument), it is necessary to propagate the string somehow. We use the `ConstantPropagation` domain of PyLiSA to achieve this by supporting the propagation of constant strings generated by assignments, operators, and methods, such as the string concat (the `+` operator) or the string constructor (`str` method).

6.3 Analysis of the `rclpy` Library

Now we are ready to present how to perform an analysis of a `rclpy` program to extract the ROS Computational Graph. In this section:

1. We want to talk about the semantics of the `rclpy` library.
2. We will discuss the configuration of the analysis.
3. We will provide a result of this analysis.

6.3.1 `rclpy` Semantics

We now present an overview of the semantics defined for some methods of the `rclpy` library. Since the analysis focuses on extracting Nodes and Topics, we considered only the methods that deal with these entities: `Node.__init__`, `Node.create_publisher()`, `Node.create_subscription()`, `Publisher.__init__`, `Subscription.__init__`.

Instantiation of a Node

Inside the class `it.unive.pylisa.library.rclpy.node.Init`, we define the semantics of a new Node. The definition is straightforward and is presented in Code [6.3](#): here, we create an `AccessInstanceGlobal` (that model a heap variable) with target (i.e., the accessed global) `"node_name"`. The receiver is

taken from the expression of the method's first parameter, that is, the Object itself. Then an assignment on this global is performed: on line 2, we assign the expression of the second parameter (corresponding to the Node's name).

```

1 AccessInstanceGlobal nodeName = new AccessInstanceGlobal(
    st.getCFG(), getLocation(), getSubExpressions()[0], "
    node_name");
2 PyAssign pyAssign = new PyAssign(getCFG(), getLocation(),
    nodeName, getSubExpressions()[1]);
3 return state.lub(pyAssign.semantics(state, interprocedural
    , expressions));

```

Code 6.3: Body of the method binarySemantics of Node Init

Instantiation of a Publisher

The semantics of the `__init__` method of the Publisher Object is defined inside `it.unive.pylisha.library.rclpy.publisher.Init` (code [6.4](#)). We assign values on `AccessInstanceGlobals`, as we did in the `Node.__init__()` semantics. In this case, however, we have multiple accesses.

```

1 AnalysisState<A,H,V,T> result = state;
2 AccessInstanceGlobal aig;
3 PyAssign pa;
4 aig = new AccessInstanceGlobal(st.getCFG(), getLocation(),
    getSubExpressions()[0], "msg_type");
5 pa = new PyAssign(getCFG(), getLocation(), aig,
    getSubExpressions()[1]);
6 result = result.lub(pa.semantics(result, interprocedural,
    expressions));
7 aig = new AccessInstanceGlobal(st.getCFG(), getLocation(),
    getSubExpressions()[0], "topic_name");
8 pa = new PyAssign(getCFG(), getLocation(), aig,
    getSubExpressions()[2]);
9 result = result.lub(pa.semantics(result, interprocedural,
    expressions));
10 aig = new AccessInstanceGlobal(st.getCFG(), getLocation(),
    getSubExpressions()[0], "qos_profile");
11 pa = new PyAssign(getCFG(), getLocation(), aig,
    getSubExpressions()[3]);

```

```

12 result = result.lub(pa.semantics(result, interprocedural,
    expressions));
13 return result;

```

Code 6.4: Body of the method expressionSemantics of Publisher Init

Instantiation of a Subscription

The semantics of the `__init__` method of a Subscription is very similar to the `Publisher.__init__()`. We add an `AccessInstanceGlobal` for the callback function and assign the corresponding expression to it. The semantics is defined inside the `it.unive.pylisa.library.rclpy.subscription.Init` class.

Create a Publisher from a Node

Inside the class `it.unive.pylisa.libraries.rclpy.node.CreatePublisher`, we define the semantics of `Node.create_publisher()` method. Here, we create a new `PyNewObj` that models the creation of a Publisher. The `PyNewObj` has the same parameters as the `create_publisher` method, without the first one (the current Node object). The LiSA type of this expression is the `Publisher ClassUnit`.

```

1 PyClassType publisherClassType = PyClassType.lookup(
    LibrarySpecificationProvider.RCLPY_PUBLISHER);
2
3 PyNewObj publisherObj = new PyNewObj(this.getCFG(), (
    SourceCodeLocation) getLocation(), "__init__",
    publisherClassType, Arrays.copyOfRange(
    getSubExpressions(), 1, getSubExpressions().length));
4 publisherObj.setOffset(st.getOffset());
5 AnalysisState<A,H,V,T> newPublisherAS = publisherObj.
    expressionSemantics(interprocedural, state, params,
    expressions);
6 return state.lub(newPublisherAS);

```

Code 6.5: Body of the method expressionSemantics of CreatePublisher

Create a Subscription from a Node

Creating a Subscription from a Node is equal to creating a Publisher, with the only difference being that the `PyNewObj` has the `Subscription ClassType`

as LiSA Type.

Putting it All Together: the PyLiSA SARD

Inside the PyLiSA SARD, we declare the Node, Publisher, and Subscription class and the rclpy library methods we use for the analysis. The snippet in code [6.6](#) shows how.

```
1 library rclpy:
2     class rclpy.publisher.Publisher:
3         instance method __init__: it.unive.pylisa.
4             libraries.rclpy.publisher.Init
5             libtype rclpy.publisher.Publisher*
6             param self type it.unive.lisa.type.Untyped::
7                 INSTANCE
8             param msg_type type it.unive.lisa.type.Untyped
9                 ::INSTANCE
10            param topic type it.unive.lisa.program.type.
11                StringType::INSTANCE
12            param qos_profile type it.unive.lisa.type.
13                Untyped::INSTANCE
14            instance method publish: it.unive.pylisa.libraries
15                .rclpy.publisher.Publish
16            type it.unive.lisa.type.VoidType::INSTANCE
17            param self libtype rclpy.publisher.Publisher*
18            param msg type it.unive.lisa.program.type.
19                StringType::INSTANCE
20
21    class rclpy.subscription.Subscription:
22        instance method __init__: it.unive.pylisa.
23            libraries.rclpy.subscription.Init
24            libtype rclpy.subscription.Subscription*
25            param self type it.unive.lisa.type.Untyped
26                ::INSTANCE
27            param msg_type type it.unive.lisa.type.
28                Untyped::INSTANCE
29            param topic type it.unive.lisa.program.
30                type.StringType::INSTANCE
31            param callback_func type it.unive.lisa.
32                type.Untyped::INSTANCE
```

```
20         param qos_profile type it.unive.lisa.type.
           Untyped::INSTANCE
21 class rclpy.node.Node:
22     instance method Node: it.unive.pylisa.libraries.
           rclpy.node.Init
23         libtype rclpy.node.Node*
24         param self libtype rclpy.node.Node*
25         param node_name type it.unive.lisa.program.
           type.StringType::INSTANCE
26     instance method __init__: it.unive.pylisa.
           libraries.rclpy.node.Init
27         libtype rclpy.node.Node*
28         param self libtype rclpy.node.Node*
29         param node_name type it.unive.lisa.program.
           type.StringType::INSTANCE
30     instance method create_publisher: it.unive.pylisa.
           libraries.rclpy.node.CreatePublisher
31         libtype rclpy.publisher.Publisher*
32         param self libtype rclpy.node.Node*
33         param msg_type type it.unive.lisa.type.Untyped
           ::INSTANCE
34         param topic type it.unive.lisa.program.type.
           StringType::INSTANCE
35         param qos_profile type it.unive.lisa.type.
           Untyped::INSTANCE
36     instance method create_subscription: it.unive.
           pylisa.libraries.rclpy.node.CreateSubscription
37         libtype rclpy.subscription.Subscription*
38         param self libtype rclpy.node.Node*
39         param msg_type type it.unive.lisa.type.Untyped
           ::INSTANCE
40         param topic type it.unive.lisa.program.type.
           StringType::INSTANCE
41         param callback type it.unive.lisa.type.Untyped
           ::INSTANCE
42         param qos_profile type it.unive.lisa.type.
           Untyped::INSTANCE
```

Code 6.6: Modeling rclpy using SARL

| COMPONENT | IMPLEMENTATION |
|---------------------------------|------------------------------|
| Interprocedural Analysis | ContextBasedAnalysis |
| Call Graph | RTACallGraph |
| Heap Domain | FieldSensitivePointBasedHeap |
| Value Domain | ConstantPropagation |
| Type Domain | InferredTypes |
| Abstract State | SimpleAbstractState |
| Semantic Checks | ROSComputationGraphDumper |

Table 6.1: LiSA configuration for the analysis

6.3.2 LiSA Analysis Configuration

Table [6.1](#) shows how LiSA is configured for the analysis.

Interprocedural Analysis

An interprocedural analysis models the effect of calls in callers and the calling context in the callees. It works like a glue that permits all the other components to work together to produce the analysis. The ContextBasedAnalysis is a context-sensitive analysis; this means that it is aware of the context where a method is called. This permits distinguishing different calls to the same method that, depending on the overall context, could produce different results.

Call Graph

We use an RTACallGraph. How it works is explained in Chapter 3, Section 5.4, which, as we said, is more efficient than a Class Hierarchy Analysis, and even if it is not one hundred percent precise, it's enough for us.

Heap Domain

We use the FieldSensitivePointBasedHeap implementation of LiSA. This heap domain abstracts heap location depending on their allocation sites (i.e., the position of the code where the heap locations are generated). This is the most precise heap domain that LiSA provides out of the box. All the field accesses, with the same field, to a specific allocation site are abstracted into a single heap identifier.

Value Domain

We use the ConstantPropagation Value Domain. This domain permits the substitution of the constant values in the expression. When we assign a constant to a variable, the domain keeps track of its value, and when the variable is used later, it is substituted with the assigned constant.

Type Domain

We use the InferredTypes Type Domain that keeps track of the inferred runtime types of an Expression by holding a set of Types.

Abstract State

The SimpleAbstractState represents the state of the analysis, and it is a composition of the FieldSensitivePointBasedHeap, ConstantPropagation, and InferredTypes. We learned about the AbstractState in Chapter 3, Section 5.2.

Semantic Checks: the ROSComputationalGraphDumper

The most interesting part of the analysis is the ROSComputationalGraphDumper semantic check.

This semantic check visits all the CFG of the program and extract all relevant information about Nodes and Topics, considering the analysis results.

When a call to a rclpy method is performed, this checker will determine what the call means, collecting useful data regarding Nodes, Publishers, and Subscribers allocations. It does this by looking at the statement under analysis. If it is a call to a rclpy Object (for example, a Node creation), it will fetch the Analysis Result of the post-state of the statement. From this, it then looks at the heap and extracts the information regarding the Object allocated: for a Subscription, for instance, it extracts the topic name, the Message Type, and the name of the Callback Function.

This information is stored in an Object that models the ROS Computational Graph. This Object is simply a set of Nodes and a Set of Topics. Whenever a new Node instantiation is detected on the CFG, a new Node is created and pushed in the ROS Computational Graph. The model of a Node is defined by its name, a ScopeId (used by identifier), a set of subscribers, and a set of publishers. A Publisher keeps track of its Node, the Topic in which the Publisher publishes, and the message type.

A Subscription has the same information as a Publisher, with the addition of

the name of the callback function.

A Topic is an Object with a single attribute: its name. When the creation of a Publisher or a Subscription is found in the CFG, the checker checks if the referred Topic is present in the ROS Computational Graph. If not, it creates it. Then, it creates a Publisher or a Subscriber, and it pushes it inside its Node.

After the execution, the ROS Computation Graph model is dumped in a .dot file for visualization. During this step, the ROS Computational Graph is analyzed to extract properties like the presence of loops (a Node that listens and publishes on the same Topic) or some oddities (for example, a Topic with zero Subscribers or zero Publishers).

6.3.3 Preliminary Results of the Analysis

For now, the analysis can take in input one file at a time. Since different files could make a ROS program, we must merge the Node definitions in a single file before analyzing the program. Merging the source code of examples [4.1](#) and [4.2](#) in a single file and performing the analysis on it, it will produce the graph in figure [6.2](#): an aquamarine circle represents a Node while a yellow rectangle is a Topic. An arrow from Node to Topic models that the Node has a Publisher on the Topic. An arrow in the inverse direction models the presence of a Subscription inside the Node that listens to messages on the Topic. Figure [6.3](#) shows the information inside the heap and value domain after creating a Publisher on the `__init__` method of `MinimalPublisher`. The banner on the left shows what the semantics defined in [6.4](#) computes: looking at the value section, we can see that we have a “msg_type” String (that corresponds to the `PyAssign` written in line 5 of [6.4](#)), a “topic_name” topic (line 8) and “qos_profile” equal to 10 (line 11). The `node_name` (`minimal_publisher`) was computed in the previous statement (that is to say, the previous node of the depicted CFG), and comes out from the semantics of the initialization of the Node (line 2 of [6.3](#)).

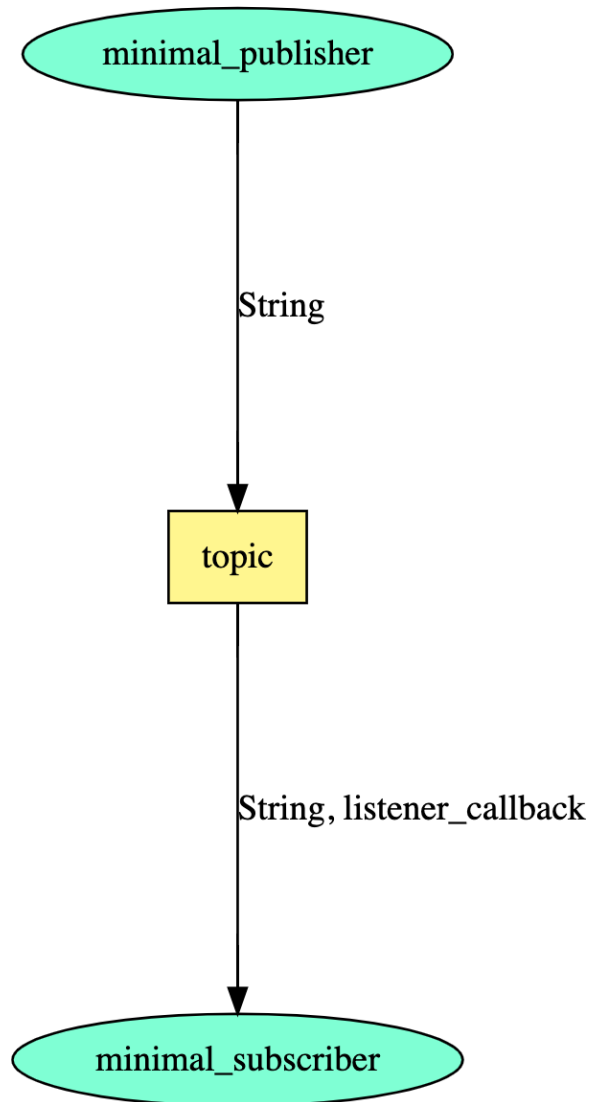


Figure 6.2: ROS Computational Graph of the Minimal Publisher and Minimal Subscriber ROS example, presented in Codes [4.2](#) and [4.1](#)

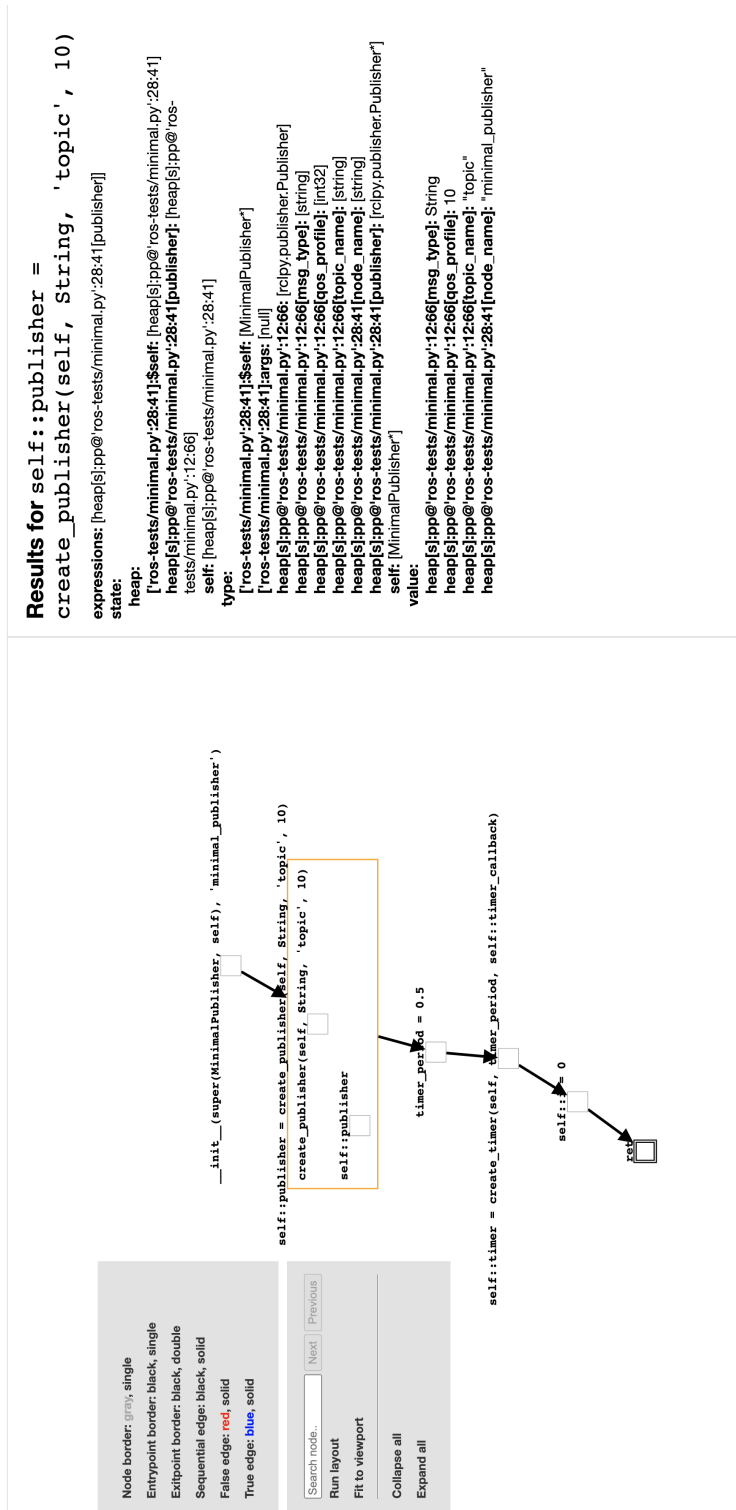


Figure 6.3: Analysis of __init__ method of MinimalPublisher

Next, let's delve into a more intricate example demonstrating how the graph displays potential errors. The graph in figure [6.4](#) represents four nodes and eight topics. As you can notice, two topics are in a different color (topic5 and topic8, in orange). These topics don't have at least one Subscription or at least one Publisher, so they could be created as an error. In addition, we have some red arrows (the arrows between node_1 and topic1 and the arrows between node_2 and topic7): this means that there is a loop (a node that publishes and reads messages on the same topics). The graph was extracted using LiSA from Code [6.7](#): as we can notice, the node_4 of figure [6.4](#) corresponds to the Node declared inside the ROSNode4 Class (lines 84-100 of [6.7](#)). Looking at the code, this Node declares two Publishers (one on topic1, line 87, and one on topic3, line 88) and a Subscription (line 89, on topic2). Red arrows mean that we have loops, and observing the code of node_1 (class ROSNode1, lines 5-27), we can see that there is both a Publisher and a Subscription on topic1 (respectively, on line 8 and line 17).

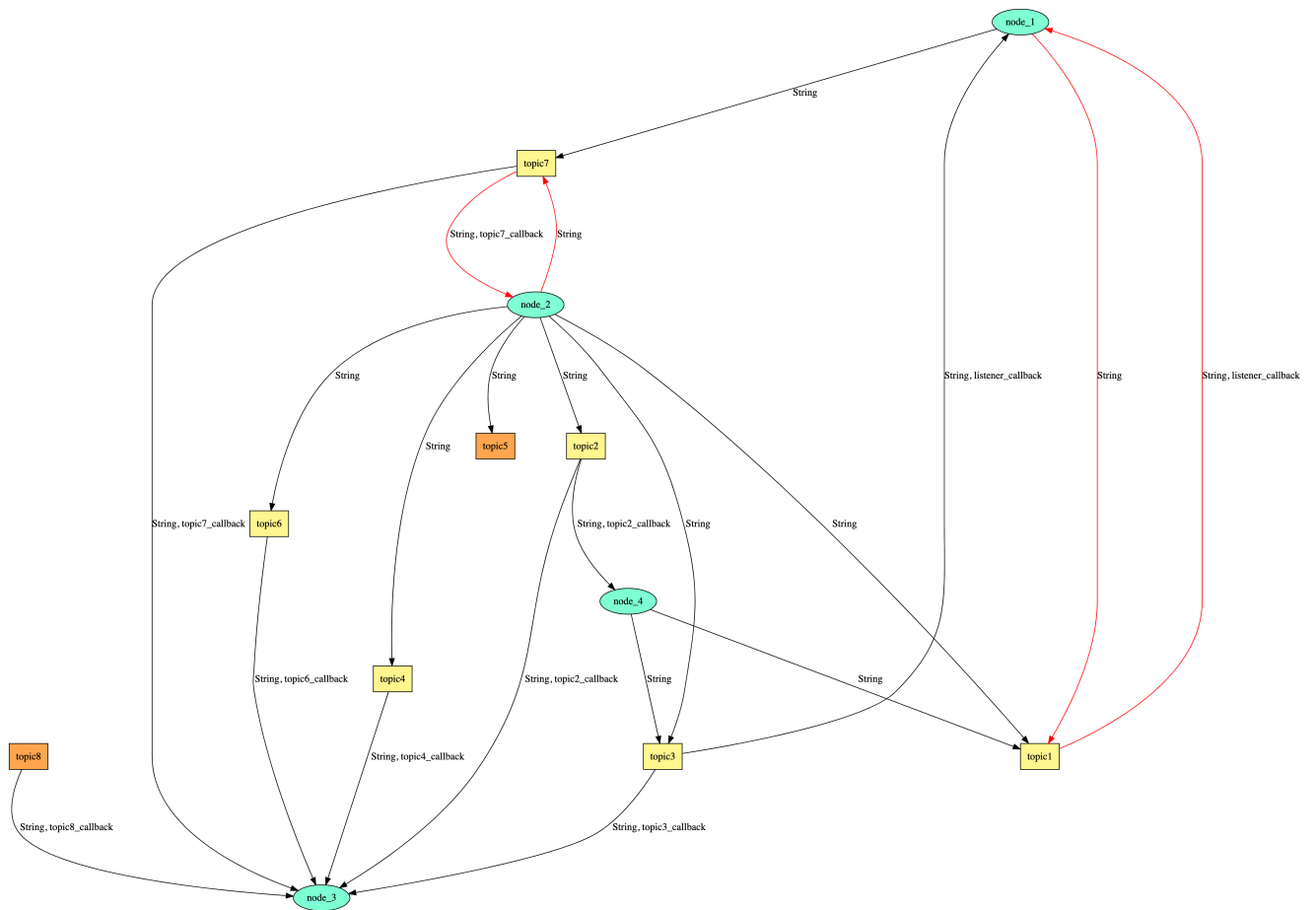


Figure 6.4: A ROS Computational Graph with four Nodes and eight Topics, extracted using LiSA from Code [6.7](#)

```
1 import rclpy
2 from rclpy.node import Node
3 from std_msgs.msg import String
4
5 class ROSNode1(Node):
6     def __init__(self):
7         super().__init__('node_1')
8         self.create_subscription(
9             String,
10            'topic1',
11            self.listener_callback,
12            10)
13        self.create_publisher(
14            String,
15            'topic7',
16            10)
17        self.create_publisher(
18            String,
19            'topic1',
20            10)
21        self.create_subscription(
22            String,
23            'topic3',
24            self.listener_callback,
25            10)
26        def listener_callback(self, msg):
27            self.get_logger().info('I heard: "%s"' % msg.data)
28
29 class ROSNode2(Node):
30     def __init__(self):
31         super().__init__('node_2')
32         self.publisher = self.create_publisher(String, '
33            topic1', 10)
34         self.create_publisher(String, 'topic2', 10)
35         self.create_publisher(String, 'topic3', 10)
36         self.create_publisher(String, 'topic4', 10)
37         self.create_publisher(String, 'topic5', 10)
38         self.create_publisher(String, 'topic6', 10)
```



```
38     self.create_publisher(String, 'topic7', 10)
39     self.create_subscription(String, 'topic7', self.
        topic7_callback, 10)
40     timer_period = 0.5 # seconds
41     self.timer = self.create_timer(timer_period, self.
        timer_callback)
42     self.i = 0
43     def topic7_callback(self, msg):
44         self.get_logger().info('I heard: "%s"' % msg.data)
45     def timer_callback(self):
46         msg = String()
47         msg.data = 'Hello World: %d' % self.i
48         self.publisher.publish(msg)
49         self.get_logger().info('Publishing: "%s"' % msg.
            data)
50         self.i += 1
51
52 class ROSNode3(Node):
53     def __init__(self):
54         super().__init__('node_3')
55         self.create_subscription(String, 'topic2', self.
            topic2_callback, 10)
56         self.create_subscription(String, 'topic3', self.
            topic3_callback, 10)
57         self.create_subscription(String, 'topic6', self.
            topic6_callback, 10)
58         self.create_subscription(String, 'topic7', self.
            topic7_callback, 10)
59         self.create_subscription(String, 'topic4', self.
            topic4_callback, 10)
60         self.create_subscription(String, 'topic8', self.
            topic8_callback, 10)
61         timer_period = 0.5 # seconds
62         self.timer = self.create_timer(timer_period, self.
            timer_callback)
63         self.i = 0
64         def topic2_callback(self, msg):
65             self.get_logger().info('I heard: "%s"' % msg.data)
66         def topic3_callback(self, msg):
```

```
67     self.get_logger().info('I heard: "%s"' % msg.data)
68     def topic6_callback(self, msg):
69         self.get_logger().info('I heard: "%s"' % msg.data)
70     def topic7_callback(self, msg):
71         self.get_logger().info('I heard: "%s"' % msg.data)
72     def topic7_callback(self, msg):
73         self.get_logger().info('I heard: "%s"' % msg.data)
74     def topic8_callback(self, msg):
75         self.get_logger().info('I heard: "%s"' % msg.data)
76     def timer_callback(self):
77         msg = String()
78         msg.data = 'Hello World: %d' % self.i
79         self.publisher.publish(msg)
80         self.get_logger().info('Publishing: "%s"' % msg.
81             data)
82         self.i += 1
83
84     class ROSNode4(Node):
85         def __init__(self):
86             super().__init__('node_4')
87             self.create_publisher(String, 'topic1', 10)
88             self.create_publisher(String, 'topic3', 10)
89             self.create_subscription(String, 'topic2', self.
90                 topic2_callback, 10)
91             timer_period = 0.5 # seconds
92             self.timer = self.create_timer(timer_period, self.
93                 timer_callback)
94             self.i = 0
95         def topic2_callback(self, msg):
96             self.get_logger().info('I heard: "%s"' % msg.data)
97         def timer_callback(self):
98             msg = String()
99             msg.data = 'Hello World: %d' % self.i
100             self.publisher.publish(msg)
101             self.get_logger().info('Publishing: "%s"' % msg.
102                 data)
103             self.i += 1
```

```
102 def main(args=None):
103     rclpy.init(args=args)
104     node1 = ROSNode1()
105     node2 = ROSNode2()
106     node3 = ROSNode3()
107     node4 = ROSNode4()
108     rclpy.shutdown()
109
110 if __name__ == '__main__':
111     main()
```

Code 6.7: Example of a ROS Source Code

Chapter 7

Future Works

Chapter 6 showed how LiSA could extract a ROS Computational Graph from a Python source code.

However, the analysis is not perfect and could be improved. Let's look at some of the improvements we can make in the future.

1. **Multi file support:** as we said in the previous chapter, the analysis can read the semantics of the application from just only one file. rclpy programs are made by more than one source. The idea is to look at the project root, fetch all the source files, and perform analysis on a set of files instead of a single one.
2. **Multiple languages support:** for some large programs, it could be the case that we have one node written in Python, for example, and another one in C++. To analyze all the nodes of a ROS program, we need to use more than one front ends that transform the source code written in the language it accepts in an intermediate LiSA program. All the LiSA programs produced by the different front ends must be merged into one single LiSA program, which is the one that LiSA will analyze. Another approach to this problem could be to treat each file as a single LiSA program, analyze them, and store all the dumped ROS Computational Graphs somewhere. Then, all the graphs must be merged into a single one. The latter approach permits avoiding recomputing the analysis for all the nodes if there is a change of the semantics for only one.
3. **Information Flow:** It could be interesting to consider an information flow analysis [35]: since Nodes exchange messages, we could see, for instance, if a top-secret message will reach an untrusted destination

that declassifies it. Denning and Denning [17] explain this problem well.

4. **Domain-Specific Domain:** the analysis used a generic Abstract State. As an improvement, we can build an ad-hoc abstract domain that models the computational graph. This adds the capability to model the intrinsic structure of a Node directly in the domain, permitting the construction of more in-depth analysis.
5. **Access Control:** if exists a Permission.xml file that specifies some policies regarding Nodes permissions, then we could parse this file and see if the dumped graph respects these policies, generating a helpful error otherwise.
6. **Actions and Services:** We could model and insert the definitions of Actions and Services in the static ROS graph.
7. **Behavior Analysis:** consider the ROS Graph represented in Figure 7.1, where we have 3 Nodes and 2 Topics. node_1 puts a string on topic1, and node_2 fetches and checks its content. If the string matches some rules, node_2 will publish a message on topic2. The node_3 reads the message from topic2 and perform some task. If node_1 did not push a particular message on topic1, node_3 (that does not have a common topic with node_1) would not have performed a specific action. So there is some dependency between node_1 and node_3, and if node_3 executes the mentioned action, it means that node_1 was in a defined state. It could be interesting to see if and how the behavior of a Node that is distant from another affects the latter's behavior.

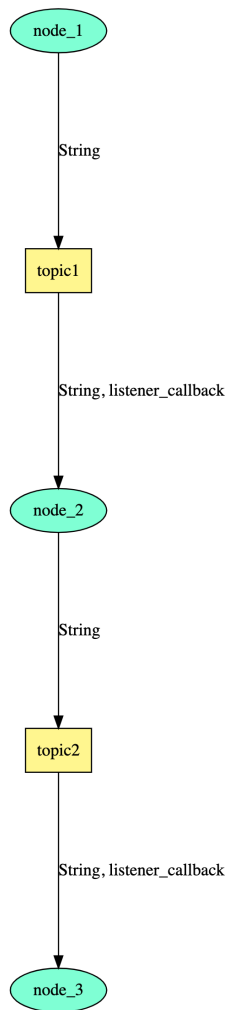


Figure 7.1: A ROS Computational Graph.

Chapter 8

Conclusion

This chapter concludes the thesis.

We learned how to generate a ROS Computational Graph at static-time using a static analysis based on Abstract Interpretation powered by the LiSA framework. We will now recap what we have seen on these pages.

In Chapter 1, Program Analysis was introduced. We talked about bugs and safety-critical systems and provided a taste of what it means (and why it is essential) to perform static analysis for robotics.

Chapter 2 set the necessary preliminaries for understanding the topics of this thesis: we discussed program languages and program analysis, with a brief overview of the main program analysis techniques (testing, model checking, static analysis), with a focus on static analysis and abstract interpretation.

In Chapter 3, we met LiSA, the defacto protagonist of this thesis. After introducing the framework, the overall internal architecture was presented with an example of how to run LiSA to perform an analysis.

After the LiSA presentation, the stage was taken by the ROS framework. We have seen how ROS works in Chapter 4, with a small example that shows how Nodes communicate with each other by using the Publisher/Subscriber mechanism.

Chapter 5 set the underlying motivation of this thesis, while in Chapter 6, we threw ourselves headlong into the analysis. We have seen how to pro-

duce a ROS Computational Graph at static-time and how the semantics of the rclpy library was modeled.

Chapter 7 explained the limitations and provided some improvements that can be made to the overall work.

In Chapter 8, we concluded the thesis, but we want to avoid going recursive and explaining the conclusion in the conclusive chapter: we will stop here.

Acknowledgements

I believe that everyone has something to teach others, and sometimes these teachings are transmitted unknowingly through exchanging words, glances, or even silence. Throughout my life, I have had the privilege of encountering extraordinary people who have imparted valuable lessons to me academically and in terms of personal growth. This thesis would not have been possible without the presence and influence of these remarkable persons who have crossed my path over the years. Therefore, I want to express my heartfelt gratitude to all of them.

Firstly, I want to thank my supervisor, Prof. Pietro Ferrara, for his patience and guidance through the development of the static analyzer. His expertise and advice have been instrumental in shaping my research. I would also like to extend my thanks to Luca Negrini, the author of LiSA, for generously dedicating his time to countless calls, helping me clarify any doubts I had regarding the library's functionality (Luca, if you are reading this, remember that I owe you some rounds of beers!). Their combined support has played a significant role in the successful completion of this project.

My thanks don't end there because I want to thank my mom and dad for always believing in me and teaching me perhaps the most important thing: never stop dreaming. I say thank you also to all my family for their support.

Lastly, but not least, I want to thank all my friends, in particular: Diletta, Sara, Gianluca, Laura, Fabrizio, Elisa, Martina, Edoardo, Marco, Salvatore, Victor, Francesco, Libero, Diego, Fabrizio, Chiara, Irene, Riccardo, and my dog Kokoro. I wish you all the best!

Cheers,
Giacomo

Bibliography

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [2] M. Albonico, M. Đorđević, E. Hamer, and I. Malavolta. Software engineering research on the robot operating system: A systematic mapping study. *Journal of Systems and Software*, 197:111574, 2023.
- [3] J. Aldrich. Object-oriented call graph construction.
- [4] M. Alhanahnah. Software quality assessment for robot operating system. 12 2020.
- [5] G. Antonelli, T. Fossen, and D. Yoerger. *Underwater Robotics*, volume 1, pages 987–1008. 01 2008.
- [6] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [7] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, 3rd edition, 1967.
- [8] A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying persistent security properties. *Comput. Lang. Syst. Struct.*, 30(3–4):231–258, oct 2004.
- [9] H. Cadavid, A. Pérez Ruiz, and C. Rocha. Reliable control architecture with plexil and ros for autonomous wheeled robots. pages 611–626, 08 2017.
- [10] G. Caiazza. *Application-level security for robotic networks*. PhD thesis, 2021.

-
- [11] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [12] A. Cortesi, P. Ferrara, and N. Chaki. Static analysis techniques for robotics software verification. 10 2013.
- [13] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, pages 505–521, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [14] P. Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.
- [15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.
- [16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 84–96, New York, NY, USA, 1978. Association for Computing Machinery.
- [17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, jul 1977.
- [18] P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 302–321, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [19] P. Ferrara. A generic framework for heap and value analyses of object-oriented programming languages. *Theoretical Computer Science*, 631:43–72, 2016.
- [20] P. Ferrara, P. Müller, and M. Novacek. Automatic inference of heap properties exploiting value domains. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 393–411, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

- [21] P. Ferrara and L. Negrini. Sarl: Oo framework specification for static analysis. In M. Christakis, N. Polikarpova, P. S. Duggirala, and P. Schrammel, editors, *Software Verification*, pages 3–20, Cham, 2020. Springer International Publishing.
- [22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999.
- [23] G. Yeap et al. 5nm cmos production technology platform featuring full-fledged euv, and high mobility channel finfets with densest 0.021 μm^2 sram cells for mobile soc and high performance computing applications. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 36.7.1–36.7.4, 2019.
- [24] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04*, page 186–197, New York, NY, USA, 2004. Association for Computing Machinery.
- [25] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '97*, page 108–124, New York, NY, USA, 1997. Association for Computing Machinery.
- [26] D. Hilbert. *Principles of Mathematical Logic*. Providence, R.I.: AMS Chelsea, 1950.
- [27] J. Kim, J. M. Smereka, C. Cheung, S. Nepal, and M. Grobler. Security and performance considerations in ros 2: A balancing act, 2018.
- [28] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), may 2022.
- [29] L. Negrini. *A generic framework for multilanguage analysis*. PhD thesis, 2023.

- [30] L. Negrini, V. Arceri, P. Ferrara, and A. Cortesi. Twinning automata and regular expressions for string static analysis, 2020.
- [31] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.
- [32] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.
- [33] J. Petereit, J. Beyerer, T. Asfour, S. Gentes, B. Hein, U. D. Hanebeck, F. Kirchner, R. Dillmann, H. H. Götting, M. Weiser, M. Gustmann, and T. Egloffstein. Robdekon: Robotic systems for decontamination in hazardous environments. In *2019 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 249–255, 2019.
- [34] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.
- [35] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming Languages and Systems, ESOP '99*, page 40–58, Berlin, Heidelberg, 1999. Springer-Verlag.
- [36] A. Santos, A. Cunha, and N. Macedo. The high-assurance ros framework. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pages 37–40, 2021.
- [37] K. Schwaber and J. Sutherland. *The scrum guide*. 2020.
- [38] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88*, page 164–174, New York, NY, USA, 1988. Association for Computing Machinery.
- [39] I. Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.
- [40] A. Sosin. How to increase the information assurance in the information age. *Journal of Defense Resources Management*, 9:45–57, 2018.
- [41] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

- [42] V. M. Vilches, R. White, G. Caiazza, and M. Arguedas. Sros2: Usable cyber security tools for ros 2, 2022.
- [43] R. Xavier and Y. Kwangkeun. *Introduction to Static Analysis - An Abstract Interpretation Perspective*. MIT Press, 2020.

