Improving Code Quality in ROS Packages Using a Temporal Extension of First-Order Logic

David Come, Julien Brunel, David Doose

To cite this version:

David Come, Julien Brunel, David Doose. Improving Code Quality in ROS Packages Using a Temporal Extension of First-Order Logic. 2018 Second IEEE International Conference on Robotic Computing (IRC), Jan 2018, Laguna Hills, France. pp.1-8, 10.1109/IRC.2018.00010 . hal-02489018

HAL Id: hal-02489018

https://hal.archives-ouvertes.fr/hal-02489018

Submitted on 24 Feb 2020

Improving code quality in ROS packages using a temporal extension of first-order logic

David Come
Universite Paul Sabatier
31400 Toulouse, France

Julien Brunel
ONERA, 2 avenue Edouard Belin
31400 Toulouse, France

David Doose
ONERA, 2 avenue Edouard Belin
31400 Toulouse, France

***Abstract*—Robots are given more and more challenging tasks in domains such as transport and delivery, farming or health. Software is key components for robots, and ROS is a popular open-source middleware for writing robotics applications. Code quality matters a lot because a poorly written software is much more likely to contain bugs and will be harder to maintain over time. Within a code base, finding faulty patterns takes a lot of time and money. We propose a framework to search automatically user-provided faulty code patterns. This framework is based on FO^{++}, a temporal extension of first-order logic, and Pangolin, a verification engine for C++ programs. We formalized with FO^{++} five faulty patterns related to ROS and embedded systems. We analyzed with Pangolin 25 ROS packages looking for occurrences of these patterns and found a total of 218 defects. To prevent the faulty patterns from arising in new ROS packages, we propose a design pattern, and we show how Pangolin can be used to enforce it.**

I. Introduction

Robotics is a growing field, and in future time, robots are expected to deploy in open and unconstrained environments with human presence. Traditionally, robots were confined within controlled environments which ensured safety. However, in a public environment with human interaction, safety is all the more important to protect people and maintain public trust.

Robot Operating System (ROS) [1] is a popular open-source middleware for robotics. It provides tools and abstractions to ease the creation of a robotic application and has bindings for several programming languages (C++, Lisp and Python). A typical ROS application consists of several basic processing units (called nodes) deployed in a certain manner and exchanging data through ROS. There are many existing components from various sources (the official website lists more than 1600 packages for the latest LTS release).

ROS is widely used in an academic context and spreads in the industry with initiatives like ROS-Industrial [2]. ROS-Industrial aims for extending ROS to provide solutions needed by the industry (such as interoperability with standard hardware, or ready to use industrial applications). ROS others drawbacks for industrial applications are the lack of real-time control and the lack of insurance on the code quality. Code quality matters a lot because a poorly written software is much more likely to contain bugs and will be harder to maintain over time [3]. This is all the more critical as robots often have a long service life.

Our goal is to find patterns in robotics software that do not respect good programming practices. Fixing these patterns (which are not necessarily bugs) will improve code quality. Finding such faulty patterns can be done manually by peer-review, but this is time and money consuming as it requires to divert one (or several) programmers from their current task to perform the review. Instead, we propose an approach in which (1) each pattern is specified in a formal language, having thus an unambiguous meaning, and (2) the detection of the pattern within the code relies on a formal technique (model checking) which is fully automatic and provides an exhaustive exploration of the code.

The specification language for patterns, which we call FO^{++}, addresses two concerns: it allows reasoning over the structure of the source code (classes, functions, inheritance relationship ...) and over properties related to the execution paths within the control flow graph (CFG) of functions (such as a constraint on the ordering of statements). We define this language as an extension of first-order logic with temporal logic. Indeed, first-order logic is suited to reasoning over structural aspects and temporal logic allows expressing properties related to the ordering of events. Our framework also includes the (automatic) detection engine for C++ code, Pangolin, which takes a pattern specification in FO^{++} and the target C++ code as inputs.

The contributions of this paper are as follows:

- We present a framework to detect patterns in source code, which includes a formal specification language, FO^{++}, and Pangolin, an engine based on model checking, which detects automatically the occurrences of patterns in source code.
- We study five suspicious code patterns that reduce code quality and apply Pangolin to detect occurrences of these patterns in 25 common ROS packages.
- We propose a design pattern for the development of new ROS packages, which prevents the problems that come with the previously identified patterns and show how to enforce this design pattern using FO^{++} and Pangolin.

The rest of the article is organized as follows. In section II, we develop on code quality in robotics. In section III and section IV, we present Pangolin and FO^{++}. In section V, we study five suspicious patterns and apply Pangolin to detect their occurrences. In section VI, we propose a design pattern for the development of new ROS packages and show how to enforce it using FO^{++} and Pangolin.

II. Robotics and code quality

A. ROS overview

ROS is a popular open-source framework providing a collection of tools to ease the development of robots, including: a set of language and platform-independent tools used for building and distributing ROS-based software; capabilities with many state-of-the-art algorithms ready to be integrated; library implementation for C++, Python, and Lisp for developing new features; a worldwide community.

A package is the smallest unit of development and release for ROS related projects. It usually provides either access to some physical devices (such as LIDAR or IMU) or the implementation of standard algorithms for robots (navigation, guidance, control, localization,...). A typical ROS application consists of nodes (processes executing different tasks) connected through a software bus provided by ROS. The nodes use a publish-subscribe architecture to exchange data. This design is flexible, promotes code reuse and simplifies complex programming.

B. Code quality on critical systems

On critical systems, safety is of the utmost importance given that lives and public trust are at stake. Thus, it is important to ensure that the code embedded in these systems is correct with respect to a specification. Two different aspects can be checked for correctness: its semantics (what it does) or its ”style” (how it is designed and written).

There are several ways to ensure that a piece of code computes the right result. Tests are commonly used to find errors and help to gain confidence in the code correctness. The code can be correct-by-construction if it is generated from models by tools that are both proven correct [4]. Finally, the code could be proven to be correct, with the help of some static analysis tools. These tools rely on methods such as abstract interpretation or deductive-methods based on Hoare logic. For instance, Frama-C with Jessie [5] is a well-known tool to verify properties in a deductive manner on a C program.

However, correctness is often not enough for software. It has to be understandable, readable and easy to update: it should be *well-written*, and one should also have confidence in it. This translates into software metrics, but also into idioms of the language to follow, domain-specific constraints, project guidelines and coding rules. For C++, see MISRA C++ [6], JSF++ [7] or HIC++ [8]. They often focus on banning or restricting C++ constructs to produce a safer subset of C++ and are widely enforced in embedded systems. The goal of enforcing them is to improve the safety and reliability of such systems. Many static analysis tools (such as Coverity or Klocwork) can do it.

.QL [9] proposes to see the code as data. It enables to run queries over a special database which contains a representation of the program, which was built using a language-specific extractor. .QL is underpinned by an untyped variant of Datalog, and unlike FO^{++}, it cannot express temporal properties over the CFG of a function.

CppDepend is another code exploration software. It allows to analyze the code structure and specify design rules to compute technical debt estimations and find code smells. It provides CQLinq (Code query over Linq [10]) to write custom queries. Again, unlike FO^{++}, it cannot express temporal properties over the CFG of a function. Besides, it does not come with a formal semantics.

Coccinelle [11] performs advanced program matching on C-code. It is underpinned by CTL-VW, a variant of Computation Tree Logic (a temporal logic) with the ability to use quantified variables over the set of expressions within a function and record their values. Its user specification language (SPatch) is close to the one used by the *diff* utility and translated into CTL-VW formula. It was successfully applied on several open sources projects such as the Linux kernel [12] and allows not only detecting patterns but also transforming the source code. Unlike Coccinelle, Pangolin targets C++ programs, and because of the object-oriented paradigm of C++, reasons about the structural aspects of the code. It also offers CTL and LTL, while Cocinelle only supports CTL-VW.

C. ROS code quality

The need for asserting the quality of ROS packages has been recently acknowledged. In [13], the authors propose *HAROS*, a framework assessing the quality of ROS packages. The goal of this framework is to ease the computation of metrics and conformance to coding standards by leveraging ROS unique features.

Moreover, enforcing methods from embedded systems on ROS packages will undoubtedly improve code quality. However, this will only fix issues that are related to C++ and that could be found in other fields (for instance, using invalid pointers). ROS has some distinctive features, and thus ROS packages should also follow some specific design rules and patterns. Currently, the ROS community has issued a coding guide [14] based on Google's C++ Style Guide and set of code metrics [15]. The coding guide mostly provides stylistic guidelines regarding naming conventions and some basic informal rules regarding code design. roslint [16] is a linter integrated within ROS, which can perform some checks.

D. Motivation

As a motivation example, we consider the use of free functions as callbacks in ROS applications. A typical ROS application is made of nodes which use a publish-subscribe mechanism to exchange data. ROS delivers the messages to the nodes with callbacks. They are functions not meant to be called directly by the developer, therefore, they should be hidden as much as possible.

Technically, within a C++ ROS program, a callback is a function that appears as the third argument of a call to `subscribe` on a `NodeHandle` variable. ROS allows both free functions and member functions to be used as callbacks. A free function can never be totally hidden. At best, it can be static or within an anonymous namespace, which prevents the developer from calling it from another file. Nevertheless, it does not prevent the developer from calling it in the current file. Moreover, a free function often couples the reception of the data with the algorithm that treats them, hindering code reusability. Thus, we want to ensure that *all callbacks are private member functions*.

Specifying a pattern that corresponds to the satisfaction (or the violation) of this rule requires reasoning about all functions, specifying that a function is member of a class, that it is private, that the third argument of a certain call is of type `NodeHandle`, etc. This is what we call the structural aspects of a pattern. Besides, if we consider for instance the violation of the rule, we also need to express that in the CFG of a certain function, there is an execution path in which at some point, there is a call to a function on a variable of type `NodeHandle`, and such that the third argument is not a private member function. Existing methods do not address well this type of analysis which combines structural aspects and properties related to execution paths within a CFG.

In this paper, we propose a framework to address this issue. It relies on FO^{++}, a temporal extension of first-order logic, and comes with Pangolin, a prototype which implements the detection of patterns through a model checking algorithm for FO^{++} formula.

III. Pangolin

Pangolin[1] is a verification engine for C++ programs. The user provides a FO^{++} specification (corresponding to the code pattern), the source code to analyze and Pangolin checks the code compliance with respect to the specification. Fig. 1 illustrates the whole process.

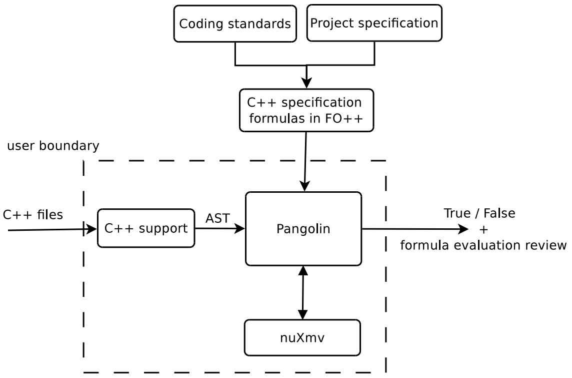

Fig. 1. Pangolin overview

Pangolin parses the C++ code source with Clang and its API libtooling [17]. Pangolin evaluates the formulas by rewriting them until it reaches true or false. It prints the result with a complete trace of the evaluation process which can be reviewed. It deals internally with most of the first order logic. It performs first-order quantifiers elimination, and evaluate structural predicates and functions. For evaluating temporal predicates, it uses the model-checker nuXmv [18] by reducing the evaluation of such predicates to a model checking problem.

Clang provides an up-to-date and complete support for C++ and direct access to the code abstract syntax tree (AST). If Clang is also used for compilation, it strengthens the analysis as the choices made about unspecified elements within the C++ standard will be the same for Pangolin and the actual execution of the program. Using Clang implies that Pangolin relies on Clang building process[2], meaning it can only analyze one file at a time. As Clang expects the code to compile, Pangolin cannot analyze a piece of code if it does not compile or a simple snippet of code taken out of its context.

In the following sub-sections, the complete construction of FO^{++} is detailed. Section IV-B defines FO^{++} syntax and then section IV-C defines its semantics. Since FO^{++} is a temporal extension of first-order logic, most of its syntax and semantics is identical to the one of first-order logic. The extension is done through parametrisation. It consists in adding two temporal predicates to the atoms of first-order logic. The semantics of these two temporal predicates follows the usual semantics of temporal logics. The details of the temporal part of FO^{++} are in section IV-D. Finally, section IV-E shows how FO^{++} is instantiated for C++ so that it can be used as a specification language for Pangolin.

IV. Specification formalism: FO^{++}

A. Overview

FO^{++} is a new logic defined as a temporal extension of first-order logic through a process called *parametrization* [19]. It has a well-defined semantics and is not tied to any programming language in its generic form. Once instantiated for a specific programming language, we use it as a specification formalism to describe faulty patterns. The first-order part of the logic is used to express structural properties.

In FO^{++}, temporal logics are used to describe a sequence of events within the CFG of a function. Traditionally, temporal logic formulas use temporal operators that allow describing sequences of events over time. In FO^{++}, an event represents a statement that can be found in the CFG of a function. Moreover, they are not sequenced with respect to time but with respect to the order in which the statements occur within the CFG.

The first-order logic is parameterized with temporal logics through two special first-order predicates $\mathrm{models}_{\mathrm{CTL}}$ and $\mathrm{models}_{\mathrm{LTL}}$. Intuitively, if f is a first-order variable and φ is an LTL formula (resp. a CTL formula) then $\mathrm{models}_{\mathrm{LTL}}(f, \varphi)$ (resp. $\mathrm{models}_{\mathrm{CTL}}(f, \varphi)$) is true if f denotes a function and if the Control Flow Graph of f satisfies the formula φ according to LTL (resp. CTL) semantics rules.

B. Syntax

a) Terms: Let V be a finite set of variables, and F^{++} a set of function symbols, each having its own arity.

We define inductively the set T^{++} of FO^{++} terms as follows:

- if x is a variable then x is a term in T^{++}

[1] Pangolin is available at https://gitlab.com/Davidbrcz/Pangolin

[2] which is the standard C++ build process

- if $f \in F^{++}$ has arity n and for each $i \in 1..n, t_i \in \mathrm{T}^{++}$ then $f(t_1, \ldots, t_n)$ is a term in T^{++}

b) Atoms: An atom consists of a call to a predicate (a predicate symbol together with a list of terms that is consistent with the profile of the predicate). The set ATOMS^{++} of FO^{++} atoms is built from the set of predicate symbols, which consists of:

- a set P^{++} of predicate symbols (each having its own arity);
- two special binary predicate symbols: $\mathrm{models_{LTL}}$ and $\mathrm{models_{CTL}}$. The first argument of both predicates is a term in T^{++}. The second argument of $\mathrm{models_{LTL}}$ (resp. $\mathrm{models_{CTL}}$) is an LTL (resp. CTL) formula as defined in section IV-D.

c) Formulas: FO^{++} formulas are defined as follows:

- $\top, \bot$ are formulas;
- if $a \in$ ATOMS^{++} then a is also a formula;
- if Q is a formula, then $\neg Q, Q \vee Q, Q \wedge Q, \forall x\ Q, \exists x\ Q, Q \Leftrightarrow Q, Q \implies Q$ are also formulas.

A valid FO^{++} formula is a formula without free-variable.

C. Semantics

a) Interpretation structure: An FO^{++} formula is interpreted over a structure $M = (D, \mathrm{CFGs}, \mathrm{has\_cfg}, \mathrm{cfg}, I, P)$, where

- D is a domain in which terms are interpreted.
- CFGs is a set of transition systems (as defined in section IV-D2) that represent the dynamic behaviors, *i.e.*, in the context of program analysis, the CFG of some elements in the domain (functions or methods). They are used to interpret temporal formulas;
- has_cfg : $D \to \{\mathrm{true}, \mathrm{false}\}$ is a total function to indicate whether a value in the domain is associated with a dynamic behavior, *i.e.*, a CFG;
- cfg : $D \to$ CFGs is a partial function, which maps some values in the domain to a CFG;
- $I \colon F^{++} \to (D^n \to D)$ defines an interpretation for functions in F^{++};
- $P \colon P^{++} \to (D^n \to \{\mathrm{true}, \mathrm{false}\})$ defines an interpretation for predicates in P^{++}.

Notice that a structure M has to satisfy the following constraint: for every element $d \in D$, has_cfg(d) if and only if d is in the domain of cfg. F^{++} and P^{++} are specific to the programming language used for the project under analysis, and D and CFGs are even specific to the program itself.

b) Environment: An environment is a partial function from the set V of variables to the domain D. If σ is an environment, x a variable in V and d a value in D, then $\sigma[x \leftarrow d]$ denotes the environment σ_1 where $\sigma_1(x) = d$ and for every $x \neq y, \sigma_1(y) = \sigma(y)$.

From an environment σ and an interpretation I for functions, we define an interpretation $K_\sigma : \mathrm{T}^{++} \to D$ for terms in the following way:

$$\begin{cases} \text{for each variable } x \text{ in } V, K_\sigma(x) = \sigma(x) \\ K_\sigma(f(t_1, \ldots, t_n)) = I(f)(K_\sigma(t_1), \ldots, K_\sigma(t_n)) \end{cases}$$

c) Satisfaction rules: Let M be a model, σ an environment and K_σ an interpretation according to this environment. We define the satisfaction relation of FO^{++} as follows[3]:

$M, K_\sigma \models \neg Q$ iff $M, K_\sigma \not\models Q$

$M, K_\sigma \models Q_1 \wedge Q_2$ iff $M, K_\sigma \models Q_1$ and $M, K_\sigma \models Q_2$

$M, K_\sigma \models \exists x\ Q$ iff

there is an $a \in D$ such that $M, K_{\sigma[x \leftarrow a]} \models Q$

$M, K_\sigma \models p(t_1, \ldots, t_1)$ iff $P(p)(K_\sigma(t_1), \ldots, K_\sigma(t_n)) = \mathrm{true}$

$M, K_\sigma \models \mathrm{models_{CTL}}(x, \psi)$ iff

$M, K_\sigma \models \mathrm{has\_cfg}(x)$ and $\mathrm{cfg}(x), K_\sigma \models_{CTL} \psi$

$M, K_\sigma \models \mathrm{models_{LTL}}(x, \psi)$ iff

$M, K_\sigma \models \mathrm{has\_cfg}(x)$ and $\mathrm{cfg}(x), K_\sigma \models_{LTL} \psi$

D. Temporal formulas

Temporal logics are formalisms usually used to describe the flow of time and how some events are ordered. Within FO^{++}, a temporal formula is evaluated over a function CFG. Thus, events are sequenced with respect to the order in which the statements occur within the CFG. Two temporal logics are available within FO^{++}: Computation-Tree Logic (CTL) and Linear Temporal Logic (LTL). LTL is discrete linear time logic, only one path is considered at a time whereas CTL is a discrete branching time. At each moment, all paths leaving the current state are considered and it offers path quantifications for temporal operators. Both are included because they are well-established logics, and tackle different issues: CTL is more natural over a CFG as it is branching-time whereas LTL offers a path-sensitive analysis and its ability to refer to the past of an event is convenient.

1) Syntax: Temporal logic are usually built on top of some fixed set of atoms (called here temporal-atoms to distinguish them from ATOMS^{++}). But here, temporal-atoms are not taken among a fixed set. Instead, they consist of calls to predicates from a set PREDCFG (disjoint from P^{++}). PREDCFG is a set of predicates which describe a fragment of AST that can be found within a function. The parameters (if any) of these predicates are terms in T^{++}, built from variables that are quantified outside the temporal predicates (we do not allow quantification in temporal formulas). Thus temporal-atoms have the following shape $q(t_1, ..., t_n)$ with $q \in$ PREDCFG$, t_1, \ldots t_n \in \mathrm{T}^{++}$.

We inductively construct LTL formulas as follows:

- temporal-atoms are valid LTL formula;
- if ψ_1, ψ_2 are valid LTL formulas, so are $\psi_1 \circ \psi_2, \neg\psi_1, \mathbf{X}\psi_1, \mathbf{G}\,\psi_1, \mathbf{F}\,\psi_1, \psi_1\ \mathbf{U}\ \psi_2, \mathbf{Y}\psi_1, \mathbf{O}\,\psi_1, \mathbf{H}\,\psi_1$ and $\psi_1\ \mathbf{S}\ \psi_2$ where $\circ$ is a binary boolean logic operator.

CTL construction is similar to LTL, expect the temporal operators are $\mathbf{A} \circ -$, $\mathbf{E} \circ -$, $\mathbf{A}[- \ \mathbf{U}\ -]$, $\mathbf{E}[- \ \mathbf{U}\ -]$ with $\circ \in \{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$.

2) Semantics:

[3] due to space constraints, we only provide the semantics of a minimal set of logical connective, which are sufficient to define the others (disjunction, implication, universal quantification, ...)

a) CFG formal definition : Let $B = (S, \to, I_{\text{CFG}}, [\![\ ]\!])$ be a CFG. S is a set of states, $\to \subseteq S \times S$ is the transition relation between states (written $\circ \to \circ$), $I_{\text{CFG}} \subseteq S$ is the set of initial states and $[\![\ ]\!] : \text{PREDCFG} \times S \to P(D^n)$ associates a predicate p of arity n with its valuation in a state s, denoted $[\![p]\!]_s$ (*i.e.*, the list of all tuples of concrete values for which the predicate is true).

b) Interpretation rules for temporal formulas: The satisfaction of LTL and CTL formulas are defined in the standard way. For the precise rules, refer for instance to [20]. In our context, the only particular satisfaction rule is for the atoms of temporal logic formulas, relying on predicates in PREDCFG. Given an environment σ, an interpretation K_σ, a predicate $p \in \text{PREDCFG}$, a state s and some terms $v_1, \ldots, v_n$, the satisfaction relation for atoms is defined as follows [4]:

$$s, K_\sigma \models p(v_1, \ldots, v_n) \text{ iff } (K_\sigma(v_1), \ldots, K_\sigma(v_n)) \in [\![p]\!]_s$$

E. FO^{++} instantiation for C++

FO^{++} construction does not mention any particular programming language and can not be used *as it is* to formalize some properties. FO^{++} needs to be instantiated for a specific programming language to be concretely used as a specification language. As the rules aim to improve the architecture of C++ ROS package, FO^{++} will be specified for C++. This process requires to instantiate the set of functions and predicates and to give them a semantics. We also have to specify precisely the concrete domain over which the formula is evaluated.

1) Domain of discourse: The domain of discourse contains the values over which FO^{++} formulas are interpreted. It is unique to each file under analysis. For a given C++ program, it contains all classes (including template ones), their attributes, member functions, constructors and destructor, all free functions, all global variables, all function arguments, and all defined types.

2) Predicates and functions: Non-temporal predicates deal with the most fundamental structural properties. In C++, structural properties are nature inquiry (*is it a class, an attribute,...*), parenthood relationship between elements, visibility, inheritance relationship, types and their qualification. For conciseness, the full list of functions and predicates is not shown here.

The meaning of FO^{++} functions and predicates corresponds to the one described in the C++ standard [21]. Table I lists some functions and predicates needed to formally express some properties in the subsequent sections and describes an informal semantics for them.

Predicate	Informal semantics
isClass(c)	true iff c refers to struct, class or union
isAttribute(c)	true iff c refers to a class/struc/union field
parent(c, p)	true iff p if the father of c
isPrivate(f)	true iff f is private within the class

TABLE I
SMALL SUBSET OF STRUCTURAL PREDICATES IN THE FO^{++} INSTANTIATION FOR C++

V. MEASURING ROS PACKAGES CONFORMANCE TO CODING RULES

In this section, we define a set of 5 coding rules that are relevant to increase the code quality within ROS packages. We then formalize with FO^{++} code patterns corresponding to the violation of the rules. Finally, we analyze 25 ROS packages with Pangolin to see if there are any occurrences of these patterns.

A. Suspicious patterns

This section shows five code patterns that can be considered bad practices for robotics systems. The first three are generic (they apply to any C++ projects on embedded systems) whereas the last two are specific to ROS. These patterns are not strictly speaking bugs but they do not convey confidence in the code. They must be at least detected, at best be fixed.

We focus on these patterns to exhibit Pangolin abilities and do not try to check the compliance of the packages with a more generic set of rules such as MISRA C++ or JSF++.

1) Generic patterns: The generic patterns address the use of global variables, variables with a scope too wide, and the use of inadequate logging mechanisms.

- R1: *All user-provided global variables must be constant* Global variables make the code harder to read and reason about for the programmer. Developers have to track the use of each variable across many lines of code to know what the code does. This is why many guidelines either reject or strictly supervise the use of global variables;
- R2: *There should be no local non-constant variable passed to a function and never used again* Variable with a scope too wide hinder code readability and can be misused in future code evolution. This is particularly visible for variables created in one scope, passed to an object or a function and never used again in the original scope. This is especially true for ROS NodeHandle as they are entry points for ROS functionality, their scope should be narrowed as much as possible. In listing 1 variables `p` and `pnh` should be attributes of that class rather than passed as arguments. Even if it is not a bug, it represents a defect in the design;
- R3: *There should be not call to* `std::cout<<`, `std::cerr<<` *in any function. No std::ofstream variables should be created.* On embedded systems, accessing resources usually follows some strict constraints in order to have guarantees on execution time and scheduling. For inputs and outputs, should not directly write on standard output and error or open custom files, all the more there is usually a dedicated mechanism for logging.

2) ROS specific patterns: For ROS specific patterns, we focus on the correct use of publishers and on callbacks.

- Each ROS publisher should be advertised and being published on
  a) R4a: *If the publisher is local to a function, then there is a call to publish within that function*

[4] it applies to both $\models_{\text{LTL}}$ and $\models_{\text{CTL}}$ and is thus simply denoted with $\models$

```
int main(int argc, char **argv){
  ros::init(argc, argv, "
      depthimage_to_laserscan");
  ros::NodeHandle n;
  ros::NodeHandle pnh("~");
  depthimage_to_laserscan::
      DepthImageToLaserScanROS dtl(n, pnh);
  ros::spin();
}
```

Listing 1. main from depthimage_to_laserscan ROS package

b) R4b: *If the publisher is an attribute, then there is a member function in which there is a call to publish on it.*

- R5: *all callbacks are private member functions*. The motivation for the rule was exposed in section II-D.

B. Pattern formalization

In this section, we show how to formally express the rules described previously. However, for conciseness, we only formalize the rule R5. We are looking for pieces of code that do not respect the rule, thus we will formally describe a code pattern that is the negation of the rule. The pattern is formally expressed with the following formula:

$$\begin{aligned}
& \exists m(\text{isFreeFunction}(m) \\
\wedge\ & \exists n(\text{locallyDeclared}(n, m) \wedge \text{hasType}(n, \textit{NodeHandle}) \\
\wedge\ & \exists c(\text{allFunctions}(c) \wedge \text{models}_{\text{CTL}}(m, \mathbf{EF}\text{sub}(n, c)) \\
\wedge\ & \neg\text{isPrivate}(c)))) \qquad (1)
\end{aligned}$$

The first part of the formula consists of finding a free function m, and then a local variable n whose type is `NodeHandle`. Then, the rest of the formula looks for a function c (that could be a free or member) and checks if it is not private. Notice that, as a free function cannot be private, the predicate isPrivate will be false if c denotes such a function. Finally, $\text{models}_{\text{CTL}}$ will be true if $\text{sub}(n, c)$ becomes finally true on, at least, one execution path in the CFG of m. `sub` is an element of PREDCFG such as $\text{sub}(n, c)$ is true on states where there is a call of the following shape $n.\textit{subscribe}(\_, \_, c)$ (c is used as the third argument of a call to `subscribe` on n).

C. Experiments

1) Corpus: We used Pangolin to find violations of the previous rules in 25 ROS packages containing a total of 172 C++ files. We chose these packages because they are well-known components and are likely to be used off the shelf for new robotics platforms. Thus, it is important to make sure their code does not contain suspicious code patterns.

The packages are sorted in three main groups:

- Navigation: all packages from ROS Navigation, extended with teb_local_planner and ros_dso (ROS Wrapper around Direct Sparse Odometry);
- Perception: a subset of ROS Perception (depthimage_to_laserscan, imu_pipeline, laser_filters, gmapping);
- LIDAR: rplidar, urg_node, loam_velodyne.

2) Results: For each formula and each file, Pangolin yields true if the formula holds on the file and false otherwise with the current value of quantified variables. To reach the end-goal of code quality improvement, when a formula is false, the user needs to review the code as there are two cases:

- a legitimate code turns out to be a counter-example for the formula. It may be because the formula was not well designed (unforeseen cases, not the intended meaning) or due to Pangolin's limitations as detailed in section III;
- the code is truly suspicious.

Although Pangolin could have analyzed all the packages from ROS last LTS release, the manual review of the results (in an effort to improve code quality) prevents us from doing so.

Rule	# files	Min	Max	Total	Nav.	Percep.	LIDAR
R1	20	1	50	179	10	5	5
R2	3	1	2	4	1	2	0
R3	3	1	2	4	1	2	0
R4a	0	0	0	0	0	0	0
R4b	6	1	2	9	5	0	1
R5	8	1	6	22	1	2	5

TABLE II
ROS QUALITY MEASURES

A summary of the results is shown table II. For each rule, the number of files in which a counter-example was found in shown in the first column. The next two columns represent the minimal and maximal number of violations found in one of the reported file. The total column is the total number of violations in all reported files. The number of violations was iteratively computed as Pangolin stops on the first counter-example it finds. The other three columns show the distribution of the files between the different groups (navigation, perception or LIDAR).

The first three rules were generic rules and not focused on ROS. Rule R1 deals with global variables and is one the simplest property. Yet, Pangolin found 20 files in which there is at least one user-provided non-constant global variable (that number drops to 10 files if we exclude test files). In the 10 remaining files, the number of global variables varies from 1 to 50. Three files have more than 40 global variables and belong to the same package: this indicates a serious design issue. This is surprising as we expected the problem of global variables to be well known and therefore their use to be limited or even absent.

Rule R2 looks for local variables used at most once and Pangolin found three files where a node handle had a scope too wide. The first one is shown listing 1. Even if the code works well, a better design would have been to make the node handlers attribute of the class. Indeed, these variables can be abused in a future development in order to quickly integrate a functionality in the code at the expense of the quality of the code. In the other two files, the node handler was created in main, and never used afterward. Thus, they pose the exact same issue.

Finally, rule R3 targets the use of general I/O instead of dedicated ones. Pangolin found three files where `std::cerr` or `std::cout` was used. Two of them were in files unrelated to ROS (main file for the Google Test Framework and one in gmapping). The last one used `std::cout` and `std::cert` before running `ros::init` to print help for command line options and report errors when parsing them. This is an example where legitimate code turns out to be a counter-example because of an unforeseen case. To take this into account, we would have to change the rule (and its corresponding pattern) to authorize the use of generic I/O in main until `ros::init` is called.

The last three look for suspicious code with ROS specific features. Even if Pangolin found no counter-examples of the property R4a, this rule might be useful during the development process to prevent bugs from copy and paste for instance.

The rule R4b is similar to rule R4a but deals with publishers which are attributes of a class. Pangolin reported 6 files in which it found a counterexample. All the publishers were actually published but not the way it was specified in the formula. These counter-examples are legitimate code that could be eliminated by lifting Pangolin limitations (such as interprocedural or multi-files analysis).

Finally, rule R5 looks for the use of free functions as callbacks. With Pangolin, we found 8 files in which at least one free function is used as a callback. Six files reported by Pangolin for this rule partly overlap with those reported for the use of global variables, suggesting a design issue. Among these six, there are the three files concentrating more than 40 global variables each.

A total of 218 defects were found (including 11 false positives), resulting in a false positive ratio of 5%. The details for each defect (rule formulation, package, and file) can be found in Pangolin's repository.

VI. The ROSApplication design pattern

In the previous section, we showed how to find suspicious code patterns within a code base with Pangolin. Yet, we can still improve code quality with finer rules to: capitalize on the ones that have proven effective; overcome the limitations of the others. To be effective, these finer rules require enforcing a specific design on the code of a package. The requirements on the design take the form of a design pattern called *ROSApplication*. Section VI-A provides an informal overview of the design pattern and then section VI-B formalizes it with FO^{++}. Notice that in the design pattern overview, we emphasis the subset of rules we will formalize in the article but the whole design pattern is available in Pangolin's repository. Finally, we perform a quick review of the design pattern.

A. Design pattern overview

The *ROSApplication* design pattern constrains the use of ROS C++ API. It is based on the rules R1, R5 of section section V and constrains the structure of the code to ensure that finer rules work properly. It also aims for:

- splitting data exchange and processing in order to have ROS agnostic algorithms;
- ensuring consistent behavior with respect to ROS communication (the mapping between publishers or subscribers and topics does not change during an execution);
- being simple to use.

```
struct ROSApplication{
  ROSApplication():rate(10){init();}
  void run(){
    while(ros::ok()){
      ros::spinOnce();
      computation();
      rate.sleep();
    }}
private:
  void init(){
    pub = nh.advertise<Msg>("pub_topic",10);
    sub = nh.subscribe("sub_topic",10,
            &ROSApplication::callback,this);
  }
  void callback(Msg const& m){/*... */ }
  void computation(){
    //...
    Msg m;
    pub.publish(m);
  }
  ros::NodeHandle nh ;
  ros::Publisher pub ;
  ros::Subscriber sub;
  ros::Rate rate     ;
};
int main(int argc, char *argv[]) {
  ros::init(argc,argv);
  ROSApplication app;
  app.run();
}
```

Listing 2. Desired architecture

Listing 2 shows an instance of the design pattern. It centralizes all ROS-related operations within the `ROSApplication` class (node handles, publishers or subscribes are forbidden in any other class or functions). Thus, analyzing this class is enough to know publishers and subscribers that are used.

To ensure a constant mapping between mapping between topics and publishers/subscribers, they should be attribute of the class. To centralize topics related operation, *there is an* `init` *method in which each publisher and subscriber is affected*. Also, *all constructors should call* `init` to ensure the publishers/subscribers are always affected.

To achieve simplicity, global variables are forbidden and callbacks are private functions. More importantly, it also provides a `run` method which acts as an event loop. Thus, main is simply reduced to the creation of a `ROSappliction` object with a call to `run` on it.

B. Formal description

For conciseness, we focus on the formalization of the rules that deal with `init`. To ease understanding, the formula is divided into several sub-formulas.

Looking for a class named `ROSApplication` is formally expressed as $\exists c\ \text{isClass}(c) \wedge \text{name}(c, \textit{ROSApplication})$.

Looking for a method named `init` is expressed as $\exists i\ \mathrm{isMemFctOf}(i, c) \wedge \mathrm{name}(i, \textit{init})$. Ensuring that the `init` method is call in all constructors is formally expressed as $\forall d\ (\mathrm{isConstructorOf}(d, c) \Rightarrow \mathrm{models}_{\mathrm{CTL}}(d, \mathbf{AF}\mathrm{call}(i)))$ where $call(f)$ is an element of PREDCFG such as it will be true on states where there is a call to f. Finally, making sure that each publisher is affected once and only once in `init` is expressed as:

$$\begin{aligned}
&\forall p(\mathrm{isAttributeOf}(p, c) \wedge \mathrm{hasType}(p, \textit{Publisher}) \Rightarrow \qquad (2)\\
&(\exists n(\mathrm{isAttributeOf}(n, c) \wedge \mathrm{hasType}(n, \textit{NodeHandle}) \wedge\\
&\mathrm{models}_{\mathrm{CTL}}(i, \mathbf{AF}(\mathrm{aPub}(p, n)) \wedge\\
&\qquad \mathbf{AG}(\mathrm{aPub}(p, n)) \Rightarrow \mathbf{AX}\ \mathbf{AG}\neg\mathrm{aPub}(p, n)))))
\end{aligned}$$

The existential quantification for the `NodeHandle` is done after the universal one for publishers because different node handles can refer to different namespaces and there is no obligation to use the same node handle for all publishers and subscribers. $\mathrm{aPub}(X, n)$ is another element of PREDCFG such as it will be true on states where there is a statement `X = n.advertise(...)`. The constraint that each element is affected at least once is expressed by the first part of the CTL formula $\mathbf{AF}\ldots$ (it should happen at least once) whereas the at most one constraint is dealt with by the second part of the CTL formula $\mathbf{AG}\ldots$ (once it has happened, it should not happen again). The formal property for subscriber attributes is similar.

C. Design pattern review

This design pattern is partially used in some of the packages we analyzed: 11 of them centralize all ROS communication within a single class. Yet, these classes are most of the time mingled with the algorithms that operate on the data. This design pattern was used to enhance ROS code quality in different packages. For instance, we re-implemented imu_bias_remover.cpp from the package imu_processors. This new implementation maintains the original code behavior while fixing the defects found using Pangolin (10 global variables, 3 free functions used as callbacks). The new code is available in Pangolin's repository.

VII. Conclusion

This paper presents a framework for checking source code compliance with an user-provided specification looking for suspicious code patterns. The framework includes a new logic called FO^{++} which is based on a parametrization of first-order logic with temporal logics. This logic allows us to conjointly specify some properties over the control flow graph of one or several functions and over the surrounding abstract syntax tree. Once instantiated for C++, it is used as a specification formalism for Pangolin, a verification engine for C++ based on Clang, and nuXmv.

To improve code quality on ROS packages, we showed how five common suspicious patterns could be formalized with FO^{++} and using Pangolin, we checked their absence in 25 ROS packages. We found several occurrences of these patterns in the packages we analyzed, some of which are real design issues. To overcome the limitations of some of the rules, we need to constrain the design of the code and thus we propose a design pattern that addresses all the issues listed before. We show how Pangolin can be used to enforce it.

There are two main directions for improving Pangolin: user interaction and expressive power. The first one would be to provide an input language closer to actual code, and to improve user feedback. The second one would be to add interprocedural and multi-file analysis. This would require extending FO^{++}'s temporal logics to distinguish in which function a statement occurs.

References

[1] M. Quigley, K. Conley, B. 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*, vol. 3, no. 3.2. Kobe, 2009.

[2] P. Evans and S. Edwards, "A disruptive community approach to industrial robotics software," RoboBusiness Leadership Summit, Tech. Rep., 2012.

[3] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 592–605.

[4] V. Rivera, N. Catano, T. Wahls, and C. Rueda, "Code generation for event-b," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 1, pp. 31–52, 2017.

[5] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International Conference on Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247.

[6] *MISRA C++: 2008: guidelines for the use of the C++ language in critical systems*. MIRA, 2008.

[7] B. Stroustrup, K. Carroll, and L. Aero, "C++ in safety-critical applications: The jsf++ coding standard," 2006.

[8] P. R. Group *et al.*, "High-integrity c++ coding standard manual," 2004.

[9] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 56, 2016.

[10] P. Pialorsi and M. Russo, *Introducing microsoft® linq*. Microsoft Press, 2007.

[11] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller, "A foundation for flow-based program matching: Using temporal logic and model checking," in *Proceedings of the 36th Symposium on Principles of Programming Languages*, ser. POPL '09. ACM, 2009, pp. 114–126.

[12] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall, "Faults in linux 2.6," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 4:1–4:40, Jun. 2014.

[13] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ros repositories," in *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2016, pp. 4491–4496.

[14] [Online]. Available: http://wiki.ros.org/CppStyleGuide

[15] [Online]. Available: http://wiki.ros.org/code_quality

[16] [Online]. Available: https://github.com/ros/roslint

[17] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014.

[18] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 334–342.

[19] C. Caleiro, C. Sernadas, and A. Sernadas, "Mechanisms for combining logics," 1999.

[20] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[21] "Iso international standard iso/iec 14882:2014(e) programming language c++."