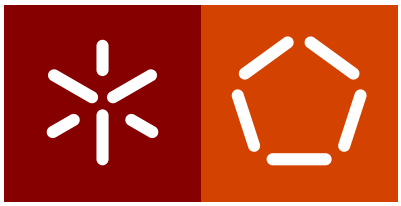
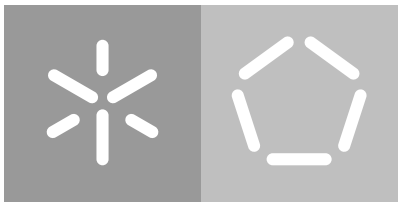


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Renato Fernandes Carvalho

Analysis of Message Passing Software Using Electrum

November 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Renato Fernandes Carvalho

Analysis of Message Passing Software Using Electrum

Master dissertation
Integrated Master's Degree in Informatics Engineering

Dissertation supervised by
Alcino Cunha
Nuno Macedo

November 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

ACKNOWLEDGEMENTS

I would like to thank to all my family and friends for supporting me in every sense during these last years of my life. Of these, I need to especially thank to Sequeira and Padrão for being there during the whole year, sharing knowledge within a friendly environment. I also need to give a special thank to Jorge, who was always available to help me with everything. I would like too to thank Andre from being always there to answer and helping me whenever I needed.

And last, but not least, I would like to thank to my Supervisors, Nuno and Alcino, for always being present and supportive with me. I am very grateful to have had the opportunity to absorb a fraction of your knowledge, within a great spirit and environment. Thank you from the bottom of my heart.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Automation developments are enabling industrial restructuring through the incorporation of more efficient and accurate processes with less associated cost. Consequently, robots are being increasingly used in the most various scenarios, including in Safety Critical domains. In such cases, the use of suitable methods to attest both the system's quality and their safety is absolutely essential.

Following the current increase of complexity of cyber-physical systems, safety guards which used to be fully hardware dependent, are constantly migrating to software. Hereupon, middleware software to abstract systems hardware are constantly evolving and are being increasingly adopted. The common feature of these systems is usually associated with its modular architectures based on message-passing communication patterns. A notorious case is the ROS middleware, where highly configurable robots are usually built by composing third-party modules. The verification of such systems is usually very hard, and its implementation in real industrial environments is, in most cases, impracticable. To promote adoption, this work advocates the use of lightweight formal methods associated with semi-automatic techniques that require minimal user input and provide valuable intuitive feedback.

This work explores and proposes a technique to automatically verify system-wide safety properties of ROS-based applications in continuous integration environments. It is based on the formalization of ROS architectural models and nodes behaviours in Electrum, a specification language of first-order temporal logic supported by a model-finder over which, system-wide properties are subsequently model-checked. In order to automate the analysis, the technique is deployed as an HAROS plug-in, a framework for quality assessment of ROS software, specially aimed to its community.

The technique proposal and its implementation under the HAROS framework are evaluated with positive results on a real agricultural robot, AgRobV16, whose dimension and complexity are industrially representative.

Keywords: Software Verification, Model Checking, Safety, Robotics, Electrum, ROS, HAROS.

RESUMO

O constante desenvolvimento em processos de automação tem motivado reestruturações nos mais diversos processos industriais, aumentando a sua eficiência, e consequentemente, reduzindo os custos associados. As vantagens provocadas pela automação impulsionam a sua adopção nos mais amplos domínios, nomeadamente, em cenários considerados críticos. Nestes casos, é vital a existência e adopção de técnicas que forneçam fortes garantias da qualidade e segurança dos sistemas.

Isto é de particular relevância aquando do desenvolvimento de sistemas ciber-físicos, onde se observa uma constante migração de *safety guards*, que eram usualmente implementadas ao nível do hardware, para lógica de software. De forma a acompanhar o aumento na complexidade destes sistemas, *middlewares* que permitem abstrair hardware têm sido adoptados de forma ubíqua. Este são construídos predominantemente sobre arquiteturas modulares baseadas em message-passing. Um caso notório são as aplicações ROS, onde robôs altamente configuráveis são construídos através da composição de módulos externos.

Na maioria dos casos, a verificação destes sistemas é muito difícil, sendo que em ambientes industriais é geralmente impraticável. Com vista a promover a adopção de técnicas que promovam a qualidade do software em ambientes de produção, este trabalho defende a utilização de *lightweight formal methods* associados a técnicas semi-automáticas que requerem intervenções mínimas por parte dos utilizadores, retornando feedback valioso de forma intuitiva.

Este trabalho explora e propõe uma técnica para verificação automática de *system-wide safety properties* em aplicações ROS, cujos resultados podem ser estendidos para qualquer arquitetura modular baseada em message passing. A técnica fundamenta-se na formalização de modelos estruturais de arquiteturas ROS, e especificações comportamentais dos seus nodos em Electrum. Após formalização do sistema, as propriedades são verificadas através de técnicas de *model-checking*. De forma a automatizar a análise, a técnica descrita neste documento é implementada através de um plug-in para HAROS, uma framework utilizada no control de qualidade de software ROS.

A técnica proposta, assim como a sua implementação sobre o ambiente Haros, foram positivamente avaliadas aquando da sua aplicação em um caso real, AgRobV16. Um robô agrícola, cuja dimensão e complexidade são representativos daquilo que seria de esperar em verdadeiros ambientes industriais.

Palavras Chave: Verificação de Software, Model Checking, Safety, Robotica, Electrum, ROS, HAROS.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

Technology developments are enabling industrial restructuring through the incorporation of more efficient and accurate processes, with less associated cost. Therefore, task automation through the use of flexible tools to assist in the most various scenarios are sought by all the spectra within the industry. Robotics is being applied in mass through a wide range of fields, including in safety-critical scenarios where system fails might jeopardize human lives.

As the complexity and systems dimensions increase, dealing with hardware-level applications tends to become impracticable. Despite the development affords, dealing with heterogeneous environment and the lack of virtualization seems to be a major issue among all the industry. Thus, modular architectures based on message-passing communication patterns are constantly appearing to address these issues.

One of the most relevant cases, is the Robot Operating System (ROS), which is a popular open source robotic framework, built to provide the needed flexibility when developing large-scale service robots. Despite the name, it is not a real operating system, but instead a middleware that provides hardware abstraction and low-level device control, which is required to develop complex software on heterogeneous environments. ROS-based applications are organized within a peer-to-peer architecture, where the system is built as a set of processing nodes cooperating with each other through message-passing models.

With the extended use of robotics and with ever-closer robot-human interaction, such as health or transportation, safety certification for robotics software are increasingly necessary. These systems usually require high level of flexibility and reliability. Safety guards that were implemented through physical ports, are gradually migrating to software logic's. Thus, the use of formal methods, especially in those domains, is advisable to avoid faults that can lead to potentially catastrophic consequences.

Methods to perform formal analysis and verification of these systems need to have a specific set of features. First, they must provide a flexible language, enabling the specification of rich structures and complex behaviour. And second, they must provide access to features that automatically check properties about the model specification.

Electrum [17] is a relevant language and framework within this context. The mixture between relational and linear temporal logic (LTL), confers to the language the capacity to express both rich structures as complex behaviour. Besides that, the toolkit that supports the

language provides automatic reasoning tools based on model-checking. These, exhaustively and automatically check whether a model meets a specific property or not, performing property verification on finite-state machines, that abstractly represent the system. The benefits in the use of model-checkers rather than state-of-art test frameworks are quite remarkable [2]. Model-checking techniques provide considerably higher degrees of coverage than conventional testing, and therefore, more reliability. As model-checking techniques act upon state-machines, there will be always a gap between the real system and the abstract one, where the verification process takes place. Thus, a suitable approach towards the analysis and verification of safety-critical properties on real systems, should start through the description of a pipeline, that considers the real system extraction to an abstract model and the analysis of some concrete pre-defined properties, and lastly, the verification results proper inspection.

Since most users of low-level programming languages have no familiarity with Formal Methods, aiming to achieve high impact in the improvement of software quality, formal techniques may be integrated through lightweight formal approaches, where completeness is sacrificed in favor of potential automation. One of the major limitations to this are usually associated with the real domain of action by the systems under analysis. Usually, for capturing relevant elements to the analysis, it's required to have high-knowledge of the execution environment and systems reversing engineering. Such processes have weak potential for automation.

During this dissertation, a novel technique to automatically verify system-wide safety properties based on Electrum, is presented. Although focused on ROS, the developed techniques and contributions are expected to be easily extended for modular message-passing based architecture. The work was guided by the following set of requirements:

- The technique must able to automatically analyse components as they are developed.
- The technique must be able to be deployed by the usual ROS software developer.
- The technique must report comprehensible feedback for all stakeholders.

Thus, the proposal of this work is to develop a novel technique to allow the verification of system-wide safety properties of message-passing based software. The technique will be based on the usage of an Electrum back-end, and it's expected to support complex multi-configuration analysis with under-specified behaviours.

To integrate the approach in the ROS development process, the analysis approach is built upon HAROS [30], a plugin-based framework for the continuous quality assessment of ROS software. The framework makes feasible the whole process by providing the required input and an integrated feedback environment. Besides that, it allows the common ROS developer to specify loose specifications of the individual nodes expected behaviour for testing purposes. Thus, the work of this thesis can be focused on the integration and

verification of ROS systems while avoiding to deal with the development of tools to extract and display information.

The structure of this document is divided into four main chapters. Chapter 2 introduces the Electurm framework and its logic, it presents a tour on the language and the analysis process through a concrete example case. Chapter 3 describes the software development in ROS, starts with the presentation of the middleware main concepts and its utility. It also approaches the quality assurance state of art in ROS, also exploring some of non-ROS relevant works. Chapter 4 presents the approach developed during this work, which allows the automatic verification of system-wide safety properties in ROS applications. Chapter 5 evaluates the usability and performance of that same approach. Lastly, Chapter 6 draws the conclusions, capturing the main limitations of using formal methods through less-formal environments, providing insights on possible future work.

2

ELECTRUM SPECIFICATION FRAMEWORK

Electrum [17] is a declarative specification language. It was forged as an Alloy [12] extension with dynamic features, loosely inspired in the Temporal Logic of Actions (*TLA*) [15]. The Alloy core provides a lightweight approach to model-based formal specifications. However, it requires the explicit modelling of dynamic behaviours, and it's presented within a framework that only supports verification through bounded model-checking [1] techniques. Thus, by extending the Alloy language through the inclusion of linear temporal logic with past operators (*PLTL*), the Electrum language merges the high-expressiveness of Alloy with a flexible form of dynamic specification as introduced in TLA [16], being therefore, well-suited to express formal state models with rich structures and complex behaviours.

To make the specification systems feasible, the Electrum language is integrated in an Electrum framework. The framework includes an IDE, to create and edit specifications, and an Analyzer capable of performing verification through bounded and unbounded model-checking techniques. The framework also includes a Visualizer that provides a graphical view of the model during each step of the modelling process. These concepts, including the proper use of the framework, will be discussed in further detail through this chapter.

2.1 LANGUAGE

The Electrum language is inspired in both Alloy and TLA, employing the former structural concepts, and the latter's ability to express and define dynamic operations.

As a language that describes software abstractions, Electrum presents itself as more than just a logic. It embodies a set of features that are commonly found in programming and modelling languages, such as polymorphism, parameterized functions and module patterns. Therefore, it enables the development and maintenance of large specifications, within a well-documented and well-known object oriented style.

Due to its complexity, the inner logic will be explained throughout the chapter. For better understanding these concepts, as those ones that will be further discussed, the Electrum formal syntax is given in the Figure 1.

```
spec ::= module qualName [ [name,+ ] ] import* paragraph*
import ::= open qualName [ [qualName,+ ] ] [ as name ]
paragraph ::= sigDecl | factDecl | funDecl | predDecl | assertDecl | checkCmd
sigDecl ::= [ var ] [ abstract ] [ mult ] sig name,+ [ sigExt ] { varDecl, *} [ block ]
sigExt ::= extends qualName | in qualName [+ qualName ]*
mult ::= lone | some | one
decl ::= [ disj ] name,+ : [disj] expr
varDecl ::= [var] decl
factDecl ::= fact [ name ] block
assertDecl ::= assert [ name ] block
funDecl ::= fun name [ [ decl, * ] ] : expr { expr }
predDecl ::= pred name [ [ decl,* ] ] block
expr ::= const | qualName | @name | this | unOp expr
  | expr binOp expr | expr arrowOp expr | expr [ expr,* ]
  | expr [ ! | not ] compareOp expr
  | expr ( => | implies ) expr else expr
  | quant decl,+ blockOrBar | ( expr ) | block
  | { decl,+ blockOrBar } | expr'
const ::= none | univ | iden
unOp ::= ! | not | no | mult | set | ~ | * | ^ | eventually | always | after | before | once
binOp ::= || | or | && | and | <=> | iff | => | implies | & | + | - | ++ | <: | :> | . |
  until | releases | since | triggered
arrowOp ::= [ mult | set ] → [ mult | set ]
compareOp :: in | =
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | | expr
quant ::= all | no | mult
checkCmd ::= check qualName [ scope ]
scope ::= for numer [ but typescope,+] | for typescope,+
typescope ::= [ exactly ] number qualName
qualName ::= [ this/ ] ( name/ )* name
```

Figure 1: Electrum's syntax.

2.1.1 *Modeling Process*

The structure specification is introduced by the declaration of *signatures*, which represent sets of uninterpreted atoms, and *fields*, that relate these atoms. If a given signature is declared as *abstract*, as in objected-oriented programming it will not have atoms beyond those within its extensions. Hierarchy can be introduced in the signatures structure through extension (keyword **`extends`**), or by the set-theory inclusion operator (keyword **`in`**). Additionally, both signatures and fields may be attached with multiplicities. In the signature case, it will constraint the number of atoms that it may contain. Whereas in fields, it will define the relation multiplicity between atoms. Moreover, it is possible to tag both, signatures and fields, as variables (keyword **`var`**). Meaning that, their evaluation may evolve through time. Otherwise, both are considered as static by default, having the same evaluation throughout every state within each given time trace.

Besides the implicit model constraints that are defined by the signature hierarchy and its imposed multiplicities, there are additional axioms (keyword **`fact`**), that can be explicitly expressed in Electrum's language. Here, every value is a relation, and every axiom that can be written is a constraint to such relations.

The Lightbot[1] is an educational video game for learning basic software programming concepts. The user's goal is to turn on all the board lights, by choosing a set of commands for the robot execution. As most of video games, there is an increasing level of difficulty that is provided by the presentation of distinct puzzles. During the following subsections, this case-study will be used to illustrate each step of a modelling and verification process in Electrum. Besides the language use exemplification, this particular case provides an insight on how to deal with variability issues. This is of great relevance, since this work frame is focused on the analysis of robotic systems, which due to the usual multi-configuration nature, are inherent characterized by high-degrees of variability.

With regard to the Lightbot software structure, there is a common set of behaviour constraints over every possible configuration, as well as a common structural part. The variability is expressed on the modelling and analysis of the different game levels (puzzles), which are pronounced by distinct board configurations. Thus, affecting only the system structure.

Common structure

In order to illustrate an Electrum *structural* specification, it follows the model depicted on Listing 2.1. The modelling of the Lightbot software might have multiple approaches. This one was built with a perception that the robot entity is placed in a unknown environment. Regardless the environment, the robot position will always be defined by a concrete location and orientation, which can vary throughout time. The environment itself can be abstractly

1 https://lightbot.com/

```
module Lightbot

enum Orientation {North, South, East, Weast}
enum State {On, Off}

abstract sig Block {
  x,y,height : one Int,
  var light : lone State,
  adj : set Block
}{
  gte[x,0] and gte[y,0]
  gte[height,0]
  adj = adjacent[this]
  light = none implies always {light = none}
}

one sig Robot {
  var orientation : one Orientation,
  var position : one Block
}

fun adjacent[b:Block] : Block→set Block {
  b →  ( ((x :> plus[b.x,1] + x :> minus[b.x,1]).Int & (y :> b.y).Int) +
         ((y :> plus[b.y,1] + y :> minus[b.y,1]).Int & (x :> b.x).Int) )
}
```

Listing 2.1: Excerpt of the Lightbot model - The structural part.

described as a maze, built as a concrete configuration of static spaced blocks. Each block may or may not have a light on it, and it's located on a two-dimensional space with a three-dimension alike attribute, that captures its height.

Therefore, the structure of this LightBot model fraction consists of:

- Some static explicit signature declarations, labeled by the keywords **sig** (`Block`, `Robot`). These will define, through a precise set of atoms, part of an immutable system configuration. The former has the **abstract** keyword, meaning that, the `Block` set will be defined only by an arbitrary number of atoms, each one within one of its possible extensions. The Block is made abstract in this first model because its extensions will be used to define a concrete table configuration. The latter is preceded by the **one** keyword, that imposes an implicit constraint over the signatures, meaning that, each model instance is required to have exactly one atom within the `Robot` set.

- Some static field declarations in the `Block` signature (`x`, `y`, `height` and `adj`). Each one defines an immutable relation between atoms in its domain signature, and its range. There are multiplicity relation constraints, which are made explicit through the use of multiplicity operators (**one**, **lone**, **set**). From those, the more uncommon operator is the

lone one, which in this particular case is used to state that, every block has at most one light `State`.

- Some variable field declarations (`light`, `orientation`, `position`). Each one defines mutable relations between atoms, attached to a specific multiplicity. These relations may evolve throughout time, as opposed to the static ones. Nevertheless, the relation is always well-defined. Relating each `Block` with any, or one of both possible `States`, namely, `On` and `Off`

- Some inner signature axioms that constraint the relations within the signature `Block` (lines 10,11,12,13). These are made by relational expressions, and some of them use built in predicates (Boolean evaluations), as `gte`, which implicitly evaluate if there is a relation of greater-than-equal between its arguments. In the expression (line 12), an auxiliary function (**fun**) named `adjacent`, is used. The use of the function is purely symbolic with readability and re-use purposes. It defines a relation between the signature itself (**this**), and its adjacent `Block` atoms, which is implicitly quantified over the whole trace. The concrete expression (lines 22,23) uses a bunch of very common Electrum operators and predicates over **Int**. Some of those are set-theory operators, such as intersection (denoted by `&`) and conjunction (denoted by `+`). Others are purely relational based operators, such as the set composition (denoted by `.`), the range explicit restriction (denoted by `:>`) and the Cartesian product operator (denoted by →). With regard to the **Int** predicates (**plus**,**minus**), they establish an obvious arithmetic relation between the arguments, that were already built in the standard language libraries. Finally, the axiom expressed in the line 14, denotes that, if a given block don't have any `light`, it must still without it during every possible time trace. Despite the rich vocabulary, through the backend, every expression is automatically combined and translated to Boolean formulas using FOL quantifiers and LTL operators.

- Some static implicit signature declarations, labeled by the keyword **enum** (`Orientation` , `State`). This is only an abbreviate form of expressing a semantic-equivalent set of statements, written in Listing 2.2. Each signature represents some entity in the game structure. In this case, there is an explicit notion of hierarchy between them. For instance, it's explicit that the abstract signature `Orientation` is composed by atoms of its sub-signatures (`North`,`South`,`East`,`West`, that represent disjoint subsets of the extended set.

Then, by running our model through a process that will be later explained, the visual diagram as shown in Figure 2 is obtained. The instance presents a non-acceptable configuration for a Lightbot puzzle, since there are configurations where two distinct blocks are placed within the same bi-dimensional coordinate.

```
1 abstract sig Orientation{}
2 one sig North extends Orientation{}
3 one sig South extends Orientation{}
4 one sig East extends Orientation{}
5 one sig Weast extends Orientation{}
6
7 abstract sig State{}
8 one sig On extends State{}
9 one sig Off extends State{}
```

Listing 2.2: The non-abreviated form of declaring an enumeration of multiple signatures that extend the abstract ones, namely Orientation and State.

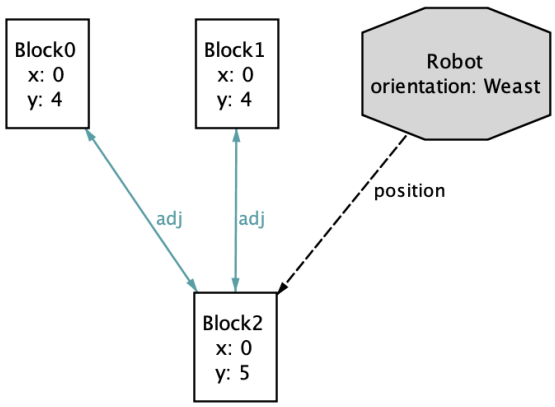


Figure 2: An inappropriate Lightbot state.

```
fact Space_Principle {
    all b1,b2: Block | (b1.x = b2.x and b1.y = b2.y) implies b1 = b2
}
```

Listing 2.3: Fact stating that, if there are two distinct blocks with the same 'x' and 'y' coordinates.

```
fact Lights_Coherence {
  all b:Block | b.light = none implies always b.light = none
                                else always b.light in (Off + On)
}
```

Listing 2.4: Fact stating that, if a given block doesn't has any light on the initial state, this will not be changed throughout the time trace. Otherwise will always be one of both, Off or On.

Thus, besides the structural properties that are implicitly considered within the signature hierarchy and fields multiplicity, there are some extra ones that should be explicitly expressed to achieve a correct model definition. One of those is the invariant property, which states that throughout every given time trace, it's always impossible to have two distinct blocks with the same (x,y) coordinates. This issue can be easily addressed by writing an explicit axiom as the one expressed in Listing 2.3.

With regard to the `light` relation, there is an axiom that must be explicitly stated to obtain a consistent model. If a given block doesn't have any light on its initial state, it must hold like that throughout every state of a given trace. This is not happening by default in the actual configuration, since the `light` relation is labeled as variable.

The Listing 2.4 addresses the issue. The novelty here is the use of the **`always`** temporal operator, whose semantics is alike an universal quantifier over time. Electrum integrates PLTL into the standard Alloy, therefore, every primitive modal operator (both unary and binary) used in linear temporal logic is available.

System Behaviour

As previously explained, the model assumptions that are not covered by the signatures structure, should be declared within Electrum facts. However, not every property that we're interested in validating and reason about, should be an assumption. Predicates are reusable boolean formulas built through Electrum expressions. All of these features, as well as the notion of reusable expressions (**`fun`**) and model assertions (**`assert`**), are conceptually grouped into Electrum paragraphs, as shown in the formal syntax (Figure 1).

Back to the Lightbot case, the actual model already defines precisely which are the valid states in the system. However, nothing has been stated regarding to its possible evolution, that is, the valid set of operations in each state.

```
1 pred move{
2     Robot.position'.height = Robot.position.height
3     Robot.orientation = North
4             implies { Robot.(position').y = plus[Robot.position.y,1]}
5     Robot.orientation = South
6             implies {Robot.(position').y = minus[Robot.position.y, 1]}
7     Robot.orientation = East
8             implies {Robot.position'.x = plus[Robot.position.x, 1]}
9     Robot.orientation = Weast
10            implies {Robot.position'.x = minus[Robot.position.x,1]}
11    (Robot.position'.x = Robot.position.x or
12            Robot.position'.y = Robot.position.y)
13    orientation' = orientation
14    light' = light
15 }
```

Listing 2.5: Specification of the `move` operation through an Electrum predicate.

Due to Electrum high-expressiveness, there are multiple forms to introduce behaviour. One of the simplest is through the declaration of implicit operations through predicates. Following the assumption that, every possible trace within the system, needs to be a random pseudo-permutation of these operations executed upon a well-defined initial state. This method, when applied consistently, represents a model design pattern known as Implicit Operation Idiom [10].

For instance, in the Lightbot software, the system state evolves, regarding to the configuration, when the robot takes some action. That is, if the robot has some command to `move`, `jump`, `turn` (left or right) or to `switch` some light (on or off). In order to express the complete possible behaviour, a predicate must be specified for each possible action. Taking the `move` operation as example, the predicate on Listing 2.5 illustrates a possible specification of the operation under an Implicit Operation Idiom.

The pre-conditions (line 2) of this operation state that, the robot position height in the next state must be the same as the actual. The pos-conditions (lines 3-11) declare the set of conditions that must to be hold in the successor state, which may vary according to the actual orientation. Finally, the frame-conditions (lines 11-14) express the conditions that depicted the remain relations, which are supposed to keep the same after the operation.

As previously claimed, in order to be possible to check and reason about properties in the system, it's necessary to introduce the notion of an execution trace. Here, upon a macro-perspective, it will be completely defined the allowed behaviour of the system, by restricting the set of valid operations to those previously stated. Thus, after every possible operation has been specified, the axiom shown in Listing 2.6 must be introduced.

```
pred init{
    Robot.orientation = North
    Robot.position.x = 0
    Robot.position.y = 0
}

fact traces {
    init
    always {
        move or
        jump or
        switch or
        turn_left or
        turn_right or
        nop
  }
}
```

Listing 2.6: Trace constraint written as an Electrum fact. The predicate init defines the robot initial position and orientation.

The `init` predicate is only used for readibility purposes. It states the set of conditions that must hold in the first state. Thereafter, every assertion about the model will be evaluated within a valid system trace.

Variability Points

The increasingly complexity of software systems tends to be gradually preceded by more efficient and robust development methods. When the systems have an indeterminate range of options and variability points, software engineering methods are applied. In this context, the systems are characterized by a common part with well-defined variability points. When this is taken into consideration, the improvements in cost and efficiency during the modelling, development and analysis of software products may be extremely substantial.

Thus, being that this work is focused on the analysis of robotic systems, which in most cases are inherently high-complex multi-configurable systems, it seems prudent to give a first approach to model variability through a less-complex system, as the Lightbot.

The Lightbot software case can be depicted and developed as a set of distinct products. There are common features that are globally used by every possible configuration and whose specification was already approached. Namely, the robot entity, its relations, the valid behaviour, and some abstract concepts that determine the kind of interface and rules that should be invariant properties of all possible configurations. Into this context, a product is built upon this common part by introducing of a concrete set of blocks arbitrarily arranged, that is, a possible puzzle.

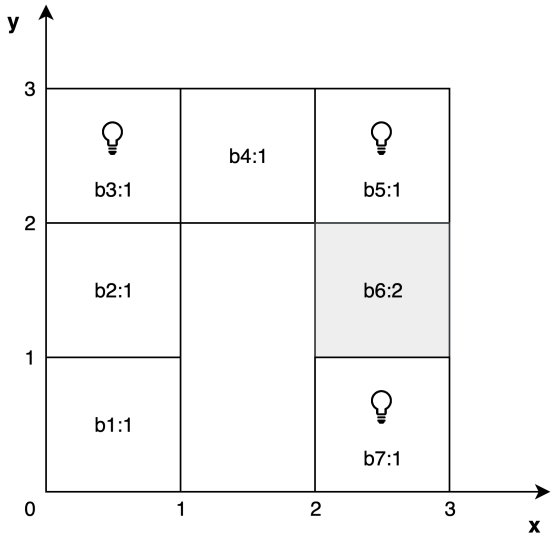


Figure 3: A graphical representation of a concrete board configuration. Each block coordinates are respective to the beginning of the block itself, and its text correspond to the (block name: height value). Each block as at most one light turn off, and distinct background colors were used to identify different height values. The white blocks correspond to height equal to one, while the grey block represents a block with height value equal to two.

The Electrum specification framework seems well-suited to deal with variability modelling and analysis. Mainly because is modular and allows the specification within multiple levels of abstraction, which seems ideal to define and reason about families of software products.

For instance, let's consider a specific puzzle configuration, portraits in Figure 3. The configuration can be modelled through simply extension of the model abstract parts.

Finally, to model a specific product as the one depicted, it's only necessary to import the library that has been previously presented, where a mixture between a common part and a meta-structure was defined. Listing 2.7 presents the specification of the puzzle, which portrays a specific product.

The sub-model structure that regards to a specific configuration consists of:

- Some static signature declarations, with a concrete multiplicity attached. Each one of those defines a concrete block, which is the building block of the table board that defines a concrete puzzle. This specification is required to be expressed at the relational level since in Electrum is impossible to write constraints at the atom's level.
- Some inner axioms to each signature, that define the specific attributes of each relation, completing the board concretization.
- An explicit fact, that defines that in the initial state, every light in the board must be turned `Off`.

```
module Lvl1
open LightBot

one sig b1 extends Block {}{
  x = 0
  y = 0
  height = 1
  light = none
}
one sig b2 extends Block {}{
  x = 0
  y = 1
  height = 1
  light = none
}
one sig b3 extends Block {}{
  x = 0
  y  = 2
  height = 1
}
one sig b4 extends Block {}{
  x = 1
  y = 2
  height = 1
  light = none
}
one sig b5 extends Block {}{
  x = 2
  y = 2
  height = 1
}
one sig b6 extends Block {}{
  x = 2
  y  = 1
  height = 2
  light = none
}
one sig b7 extends Block {}{
  x = 2
  y = 0
  height = 1
}

fact lights {
    Block.light in Off
}
```

Listing 2.7: A concrete product specification, that corresponds to the Lightbot software game with the puzzle depicted in the Figure 3

2.2 ANALYSIS

In order to endorse our intuition towards verification, or to reveal subtle flaws from scenarios that have not been exploit, analysis may encourage us to explore, depicting our current system configuration and behaviour through concrete examples.

This section will give a brief overview of the Electrum analysis system, describing the commands that are provided by the language, as well as, how they are integrated in the analysis process within the Electrum Analyzer. Lastly, to better understand the differences between the available Model-Checking techniques, as well as how they complement themselves in the analysis process, a further description on Bounded and Unbounded Model-Checking is given.

2.2.1 *Commands and Scopes*

The verification commands used to verify the model properties are integrated in the specification file. As previously shown in Figure 1, there are **`check`** and **`run`** commands. The **`check`** command is evoked with an assertion (the property), in which case the model checker will produce two formulas, that respectively characterize the model (from the declarations and facts) (φ_M) and its property (φ_P). Thus, the Electrum tries to find out if the formula ($\varphi_M \implies \varphi_P$) is valid within the scopes, which are the maximum bounds for the number of atoms within each declared signature. As will be discussed later, this will be reduced to a SAT problem. Being that, this will be translated to find out if ($\varphi_M \wedge \neg\varphi_P$) is satisfied. Regarding **`run`** commands, the model-checker is instructed to yield a concrete example of the specification, reducing the problem to the satisfiability of the formula ($\varphi_M \wedge \varphi_P$). Since there are no decidable logics, which are rich enough to capture software abstractions, Electrum logics is no exception and therefore, is undecidable. Thus, to make instance finding feasible, as said before, the commands are always specified along with the signature scopes. These should be explicitly specified for each signature, otherwise, they will assume a standard default value.

Besides that, the maximum number of different time instants are defined in the command as well, through a special **`Time`** scope. As will be further discussed, for both commands there are two possible model-checking techniques that can be used, Bounded or Unbounded. The former will use the **`Time`** bound, while the latter will simply ignore it.

In the Listing 2.8, referring to the Lightbot case, a **`run`** command labeled as `try_solution` is used. Besides the expression, which will be further explained, it is relevant to notice how the scope is specified along the command. In this particular case, none unless the Time scope is explicitly declared. However, a general scope of four is defined. This means that, for each

```
run try_solution{
  move;
  move;
  turn_right;
  switch;
  move;
  move;
  turn_right;
  switch;
  jump;
  jump;
  switch;
  always nop
}  for 4 but 12 Time
```

Listing 2.8: Exemplification of a possible time trace that may solve the puzzle presented on the Figure 3.

signature over which there is no contradictory implicit multiplicity constraint, the maximum number of atoms that will contain, is four.

2.2.2 *Electrum Analyzer*

While analyzing an abstract system design, it's expected to reason about properties that are conceptually grouped within one of two distinct categories. One of those groups is known as safety properties, which state that nothing bad happens, meaning that the system will not step into an unexpected state during its execution. The other group, is known as liveness properties, which state that something good will happen, meaning that an expected state will eventually be reached during the system execution.

The specification process is usually performed interactively. The user takes a first approach to the model specification, and then proceed to its refinement towards the model validation. In order to ease this process and improve the user comprehension of model instances and counter-examples, the Electrum Analyzer provides a Visualizer capable of showing a navigable set of graphical instances that are produced by command executions. A set of features that allow a full instance theme customization and an instant expression evaluation are also provided by the Analyzer.

In order to better illustrate how these features are integrated and combined in the analysis process, lets recover the Lightbot case. Considering the actual specification of the concrete puzzle, which has been shown in the Figure 3, there are some properties expected to be held.

Producing a concrete instance may be quite useful during the modeling process. The Listing 2.8 presented above, shows a possible use of the **run** command for this purpose.

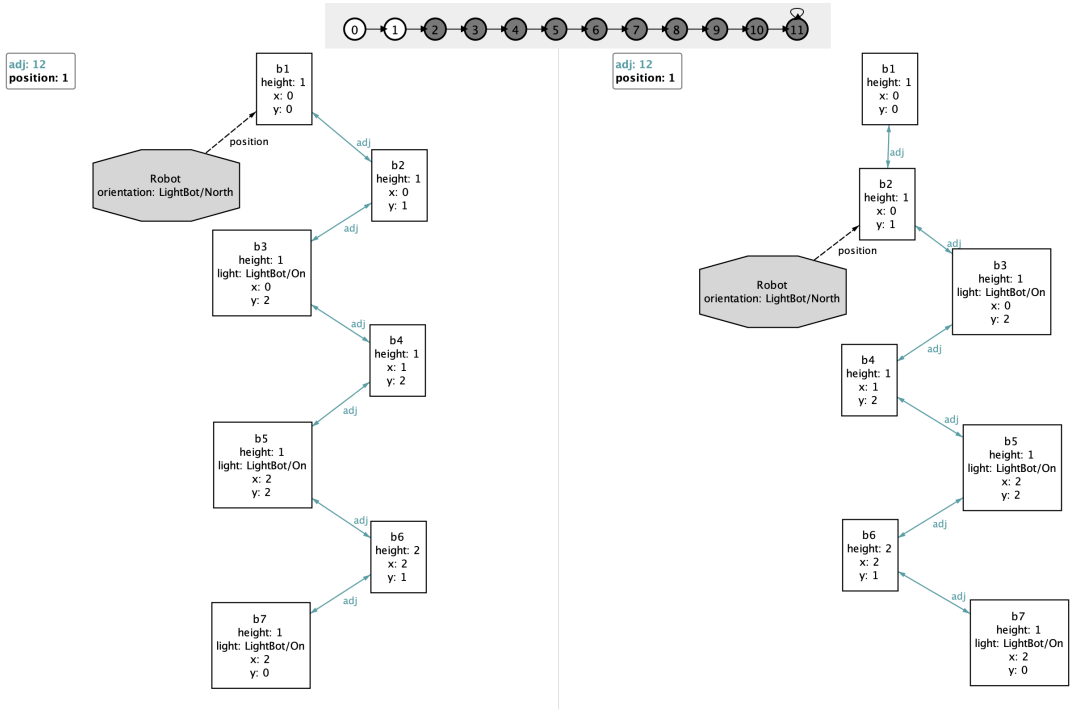


Figure 4: Partial graphical view of the two initial states. Adjacent instance states are always presented side by side. The initial state is shown on the left side, the second on the right side.

```
1 assert reachability{
2   always (all b: Block | b in Robot.position.^adj)
3 }
4 check reachability for 4 but 10 Time
```

Listing 2.9: Specification and bounded verification of the reachability assertion.

The definition introduces a very useful operator (denoted by ';'), that allows the creation of traces by operation chaining. Finally, after the command has been executed, the analyzer generates a concrete instance, and produce its visual form. In the Figure 4 is shown how the time trace is displayed.

Although using the **run** command might be useful for obtaining feedback about the model specification, it is not intended to be used for verification. Instead, the **check** command is used. In the Listing 2.9, a safety property that should be held by the system is specified.

The property states that, regardless of the robot position, every block is always reachable. The positive transitive closure operator (denoted by ^), of the binary relation `adj`, is the smallest non-empty transitive relation containing the relation `adj`. The following definition may be easier to illustrate the set that is produced by the relation:

```
^adj = adj + adj.adj + adj.adj.adj + ....
```

Notice that, this definition is possible to be defined due to the finitude of the universe in Electrum. As expected, after executing the **`check`** command upon the asserted property, the Analyzer indicates that there are no counter-examples within the specified scope. Due to the Electrum translation into LTL, every considered instance encodes an infinite trace through looping states, both in bounded and unbounded model checking cases. For the sake of performance, the Bounded Model Checking technique is usually the first approach towards the model validation. However is not complete, since the assertion is only checked within traces with a finite number of distinct states. These are limited to the **`Time`** specified scope. Even so, instance finding through the use of SAT solvers and abstract models have far more extensive coverage than traditional testing methods. And besides that, assuming the veracity of the small scope hypothesis statement:

"Most bugs have small counterexamples" [12]

One can expect to have a good approach to a valid model once every property that is expected to hold is checked, and no counter-examples were found. Then, when confident that the specification is correct, unbounded model-checking techniques can be used through the analyzer towards the model verification.

2.2.3 *Model-Checking*

Model checking is the automatic process towards the verification if a given specification meets the system finite-state defined by a concrete model.

In Electrum, unlike other analysis tools, there is a non-clear distinction between the model and the properties specification. By introduction of concrete abstraction processes and logic translations, the relational model specifications may be converted to model-checking problems. There are two main categories of model-checkers that are included in the Electrum framework. Bounded model-checkers, which only consider computational traces with a limit number of distinct states, and unbounded ones, that discard this constraint.

As previously referred, on the enunciation of the *small scope hypothesis*, bounded model-checkers are ideal to be used in the early states of analysis. They are able to produce quick feedback's, and the majority of bugs are identified through small counter-examples. However, although bounded model-checking techniques can achieve much higher coverage levels than standard testing, they are still not complete, covering only an *infimus* part of every possible case. Thus, after the model under analysis has been validated, unbounded model-checkers can complement the analysis process by offering high-levels of reliability, towards verification.

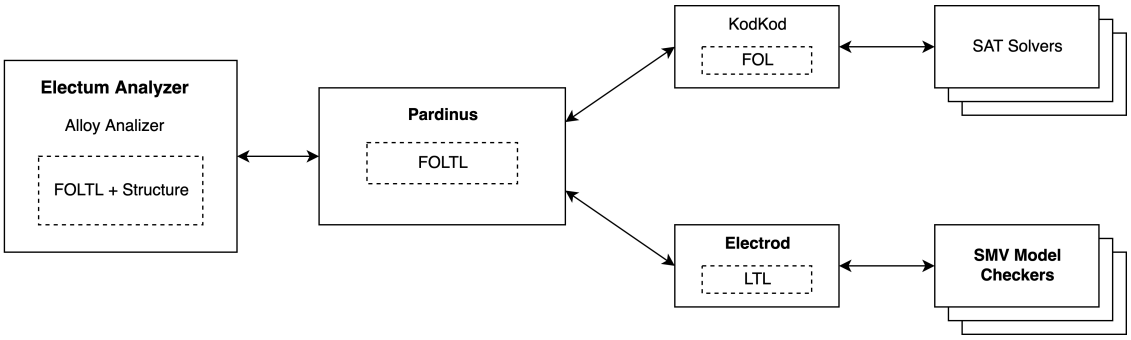


Figure 5: Electrum Architecture components. The Pardinus layer complements the Alloy KodKod with temporal logic interpretation. The Electrod component provides support to the symbolic model checkers [19].

To enable the use of both techniques, the Electrum architecture (Figure 5) was built following the same principles as in Alloy. The Electrum Analyzer is implemented on top of the Alloy Analyzer, and relies on the Pardinus model-finder, which was built upon KodKod [36].

An Electrum specification might be seen as FOLTL [3][14] logic's with structure. However, the Pardinus only uses FOLTL logic's. Thus, a transformation is required. The main operation consists on removing all relational terms present in the Electrum Kernel formulas, replacing them with the corresponding FOLTL sub-formulas. When the interpretation is complete, the Pardinus has two distinct options. One is to practice model-checking through SAT solvers [6], while the other is to do it upon symbolic based model checkers (SMV [20]). The former performs bounded-verification, while the latter supports both, bounded or unbounded verification.

If the option is to execute model-checking through SAT, the traces must be made explicit. Thus, the FOLTL logic's are expanded into FOL, and the Kodkod is used to interface with multiple SAT-solvers. When the aim is to execute model-checking upon SMV, the FOLTL logic's are expanded into LTL, for later use of Electrod[2]. This tool compiles the LTL problems into SMV.

In this work context it's not relevant to understand the intermediates logic's itself, or how they are converted between components. Thus, the following sub-sections will give a major focus on the Electrum options to perform verification. Namely, through bounded and unbounded model-checking techniques.

Bounded model-checking

The conventional way to perform bounded model-checking is based on the use of SAT solvers. The Electrum simply does the translation of its specifications into FOLTL formulas, that can be directly encoded into Alloy. This translation is feasible by explicitly introducing

2 Stand-alone tool available under the MPL 2.0 license at https://github.com/grayswandyr/electrod.

the time signature with a total order forced on it. In bounded model-checking using SAT solvers, the Boolean formula is satisfiable, if the underlying encoded state transition system can compute a finite sequence of state transitions, towards a given goal state. If the path segment cannot be found at a given length, the search continues through incremental larger ones. When checking some property that doesn't hold, the Electrum Analyzer yields a first counter example. The user can iterate allowing over all the different possible ones. This is achieved by simply re-running a conjunction of the original SAT problem, with the negated formulas that depicts the instances already produced. This is made efficient due to the use of incremental solvers that don't need to reboot the process. Notice that, each instance that was found, is constrained to a maximum number of time instants, which value was specified as the **`Time`** scope.

Unlike in theorem proving, this kind of analysis is not complete, since it only examines a finite set of cases. However, due to the recent advances in constraint solving technologies, namely in SAT solvers, the amount of cases that can be examined within a small amount of time is huge (billion order), and therefore, a percentage of coverage that it's impossible to obtain through testing.

Unbounded model-checking

The Electrum unbounded model-checking technique is implemented through the embodiment of the SMV-based tools, including NuSMV and NuXmv [5]. These equipped Electrum with a series of algorithms that efficiently perform bounded and unbounded model-checking.

The incorporation of this tool into the analyzer is performed through a chain of model translations. The model is translated to a FOLTL version, following to an LTL one. After the conversion process is completed, the SMV version of the original Electrum model is obtained. The LTL formula within the Electrum specification is translated as well. The initial explicit transition system disappears, signatures and fields are converted to frozen or plain variables, depending on they being static or variables. Formulas that were related with inclusion of signatures and fields are combined as invariants, this allows to constrain and to reduce the possible state space. Finally, the property under test is specified as an LTL formula as a SMV specification, allowing the SMV-based tool to proceed with the verification.

Notice that, although this feature allows a possible verification of properties under unbounded time constraints, it does not allow the performing of scenario exploration by iterating over counter examples, as it's done by the bounded-model check technique based on SAT.

3

SOFTWARE DEVELOPMENT IN ROS

As the scale and scope of the robotics grows, the robotic software is being increasingly used in the most diverse scenarios, from simple assisting on basic processes automatization, up to full performance of critical tasks. Besides that, the growth in used domains is followed by increases in complexity. Safety controllers are moving completely towards the software level, replacing the previous safety guards which used to be fully hardware-implemented.

Writing robotic software is generally a difficult task. Different types of robots can vary widely in their hardware as in fields of action. The reuse of code is non-trivial, and therefore, the development in large scale is made unsustainable. In order to address these problems, the Robotic Operation System (*ROS*) [28][23] presents itself as a middleware system, designed to ease the development of robotic systems in large scale.

This chapter presents a detailed introduction to the ROS architecture and its basic concepts. Thereafter, it will follow a full section dedicated to review the state-of-art on the Quality Assurance (*QA*) of robotic software, namely regarding to ROS systems.

3.1 ARCHITECTURE AND CONCEPTS

Essentially, ROS is a middleware that provides management services for heterogeneous computer clusters, such as hardware abstraction, low-level device control, and core functionalities as the message-passing between processes. It is displayed as a distributed layer between the top application layer, the operation systems and its communication facilities.

Its architecture is based on a hybrid peer-to-peer implementation, where an arbitrary number of computation nodes can cooperate with each other by communicating through message-passing patterns. The non-necessity to hold state on a central server, and the primarily use of publish/subscribe communication type, confers fault-tolerance features to these systems, enabling its use in heterogeneous environments to compute and communicate.

For instance, one of the standard problems that have inspirit the ROS design was commonly referred to as *fetch an item* problem [29]. In this case, a relatively large and complex robot is equipped with cameras, several scanners, a manipulator arm, and a wheeler base. The robot tries to find a requested item and deliver it on a default location. Usually, this kind

of systems has some architecture organization within groups. Heavy processing tasks are usually delivered to high-performance machines dedicated to graphical computation, while less-intensive tasks are attributed to less-expensive robot components. This naturally leads to an architecture representation as a graph.

The computation graph of a ROS application is the network of named resources, that process and share data. These processes are runtime entities, namely *Node Instances*, *Topics*, *Parameters* and *Services*, which will be further discussed during this section.

3.1.1 *Nodes and Nodelets*

The software written in ROS is organized in packages. Packages might contain ROS nodes, independent libraries, data-sets, configuration files, or something else that logically constitutes a useful module. The Node is the minimal building block of a ROS application. It can be seen as a standard computer process within a ROS package. In a ROS application, each node is identified by an unique name, enabling an unambiguous communication between them. The communication is mostly happens under message-passing models, which confers fault tolerance features to ROS applications, since crashes are isolated to individual nodes. Besides that, it naturally modules the code and its functionalities. This provides a well-suited environment towards the development of complex robotic software systems in large scale.

Beside nodes, that in some sense, may be seen as operative system processes, there are nodelets. These are another type of computation in ROS that can induce performance improvements over the entire system. Nodelets specifically aggregate multiple primal nodes computation, as computer threads. Which might be useful to group lightweight processes that constantly transfer high volumes of data between each other, preventing network overloading and increasing transfer rates.

A node can be implemented in any programming language, as long as there is binding for ROS. However, the most common is the use of C++ or Python, due to the support focus. This is made by providing ROS client libraries such as[1]:

- *roscpp*: A C++ client library for ROS. It the most used ROS client library, and was designed to be the high performance library for ROS.
- *rospy*: The Python client library for ROS. It was designed to provide the advantages of an object oriented scripting language to ROS. The design of rospy takes advantage of the high-level language features, easing algorithm prototyping with inexpensive configuration efforts, that are counterbalanced through performance costs.

Through the use of these libraries, a programmer can easily assess and make use of the most relevant ROS resources. For exemplification purposes, the Listing 3.1 shows a dummy

1 http://wiki.ros.org/Client Libraries

```
1 #include "ros/ros.h"
2 #include "std_msgs/Int64.h"
3 #include <stdlib.h>
4
5 int main(int argc, char **argv){
6     ros::init(argc,argv,"dummy_sensor");
7     ros::NodeHandle n;
8     ros::Publicher pub = n.advertise<std_msgs::Int64>("sensor_data", 10);
9     ros::Rate loop_rate(10);
10
11     std_msgs:Int64 msg;
12     while(ros::ok()){
13         int value = rand() % 100 + 1;
14         msg.data = value;
15         pub.publish(msg);
16         loop_rate.sleep();
17     }
18
19     return 0;
20 }
```

Listing 3.1: A C++ node definition. It tries to mimic a sensor behaviour, which publishes random values, between 1 and 100, on a given topic.

implementation of a ROS sensor emulator, that constantly generates integer random values smaller than 100.

Aiming a better comprehension of this simple specification, the code can be broken into two structural parts.

- The lines 1-3 are used to import libraries. The first to import the correspondent roscpp library. The second and the third, to import external libraries that will be used in this specification.

- The lines 5-16 are used to specify an emulation of a sensor behaviour. The first and second roscpp calls (`init`, **`advertise`**) are used to declare that the node will be initiated with the base name `dummy_sensor`, and that will be publishing data on the `sensor_data` topic 3.1.2 using the message queue size specified as its second argument. Any future use of this resource entity within the system, is made through this resource name. With regard to the `while` cycle, it will be executed until the system stops, at a given frequency of 10Hz. It will simply proceed in each interaction, with the generation of a pseudo-random positive value (1-100), following with its publication on the `sensor_data` topic.

This code sample will be specially relevant in further sections along the chapter. The used concepts that were not properly explained, as the *topic* use, as well as the different forms of communication between nodes, will be further explained.

ROS Master

As said before, one of the main goals of ROS is to enable developers to design large scale systems as collections of small independent programs. In order to develop complex systems, the set of nodes need to be cooperative. The service that provides the necessary information in order to enable nodes to communicate within a peer-to-peer context, is known as *roscore*[2]. This service starts a node (Ros Master) that is globally known by the entire system. When all the other nodes have started, immediately connects to roscore, passing it the register details with its entire description, as its name and topics that it intends to publish/subscribe on. The information request about services that they intend to use is also passed to the Ros Master.

Thus, the roscore node provides registration services, resource localization and service lookup to the entire system. Lastly, it also provides a *parameter server*, that is usually used in nodes configuration. Basically, it allows nodes to store arbitrary data, such as configuration variables or algorithms parameters, to be used at runtime.

3.1.2 *Communication*

In ROS concepts, a message is a well-typed data structure, that by default is built upon some set of ROS primitive types. These primitive types might include Floating Points (*float32* or *float64*), Booleans (*bool*), Integers (*int8*, *int16* or *int64*), among less-common others. Besides that, basic constructions as arrays of primitive types and constants are also supported.

The communication between nodes happens essentially through message passing models. The concrete mechanisms that are used to pass message through topics will be discussed throughout this section. Is important to notice that, every peer-to-peer connection and negotiation occurs in XML-RPC, for which there are reasonable implementations in most major programming languages. Thus, the concepts and details that are being discussed are completely language-independent.

Topics

Topics are by far the most used form of communication in ROS systems. They are essentially named buses over which nodes may exchange messages. A Publish/Subscribe semantic is used to decouple the information production and its consumption. Through the development of large-scale software within an heterogeneous environment, this communication pattern provides all sort of advantages [9][35], such as:

- Separation of concerns: Since most robotic software that is developed through ROS is modular, both in code and in functionality. By using this communication model, it's

2 https://wiki.ros.org/roscore

possible to obviate the inherent iteration complexity between modules, enabling the development of these systems in large scale.

- Improved testability: Due to the high control over topics that is achievable, it is easy to test if the event buses are sending the correct messages.
- Reduced cognitive load for subscribers: From the subscriber perspective, the publisher only exists as a black box. Therefore, subscribers don't need to consider the publisher work in its definition, being able to delay the message consumption as much as they need.
- Scalability: This model provides better scalability through message caching, parallel operations and default efficiently routing options.
- Improved Security and Fault Tolerance: This communication architecture rests on the principle of assigning minimal privileges to its entities. Also, if some node publishes, or subscribes some topic that is discarded, it won't directly affect the rest of the system components, since they are not aware of each others by build principle.

When some node intends to publish a message on a given Topic, it connects with *roscore* to advertise it. Which, in its turn, is responsible to alert each subscriber with the publisher details, enabling message passing between them. With the aim of better illustrating this process, Listing 3.2 specifies a node that's function is to receive data from the dummy sensor (Listing 3.1), processing it, producing the results to a second topic. This example can be interpreted as a micro robotic system, where the sensor captures data from the environment and communicates with a safety node. This node will interpret the received data, computing a velocity that will be publish to a given topic. This topic may be used, for instance, by an arbitrary robotic actuator drive, whose definition is unknown.

The safety node implementation is pretty straightforward. The **`subscribe`** function is used to receive data from the `sensor_data`. Its second parameter is the size of the message queue. If messages are arriving faster than they are being processed, this number determines how many messages will be buffered up before beginning to throw away the oldest ones. The third and last **`subscribe`** parameter indicates the callback function that should process the received messages.

The single peculiarity in this specification is the use of the roscpp `spinOnce` method (line 27). This method calls once all the callbacks waiting to be called at that point in time. There are a multiple `spin` functions that can be used. However, this is simplest one, being specially meant for single-thread applications. When a given message arrives, the thread will execute the callback function which is associated to the respective topic. The use of a global variable `vel` to carry results, is far from being a good practice, however, being that the `spinOnce` function is single-threaded, possible concurrency issues can be discharged.

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

int vel;

void sensorCallback(const std_msgs::Int64::ConstPtr msg){
    int value = msg.data;
    if (value > 10){
        vel = 5;
    }
    else{
        vel = 0;
    }
}

int main(int argc, char **argv){
    ros::init(argc,argv,"safety_node");
    ros::NodeHandler n;
    ros::Subscriber sub = n.subscribe("sensor_data",1000,sensorCallback);
    ros::Publisher pub = n.advertise<std_msgs::String>("safe_vel",10);
    ros::Rate loop_rate(10);
    std_msgs::Int64 msg;

    while(ros::ok()){
        msg.data = vel;
        pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

Listing 3.2: The safety node implementation in ROS.

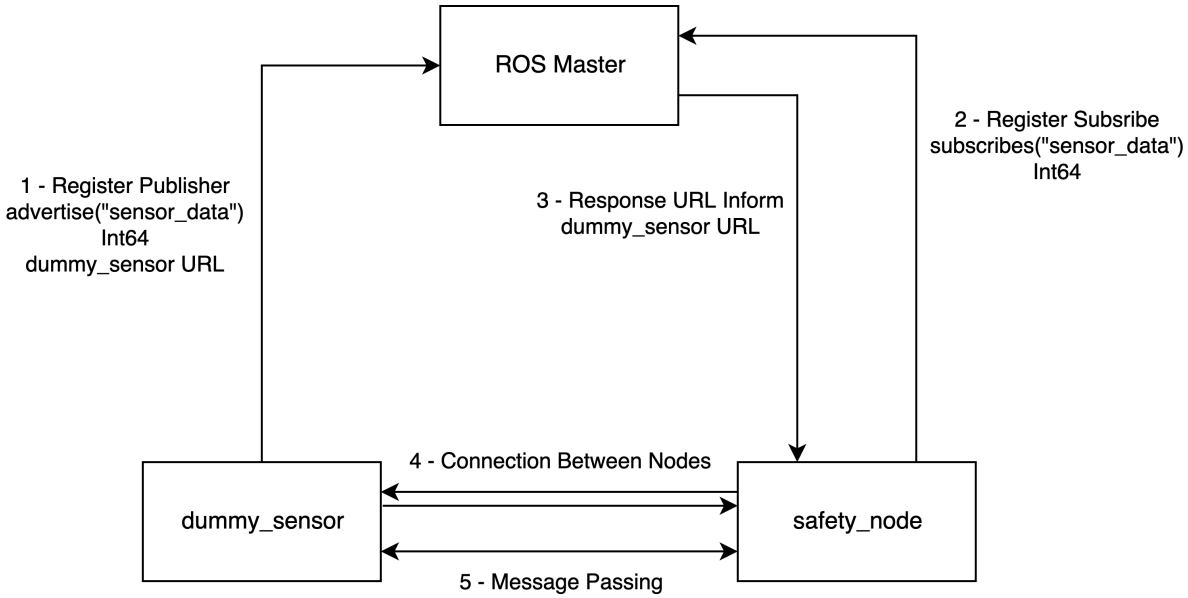


Figure 6: The advertise/subscribe process on the `sensor_data` topic. Each node represents a process. The links are labeled with a sequence number, representing the communication performed between them. Notice that, up to the (4) step, every communication is made through XML-RPC. After that, a TCP link is established.

Going back to the main point of this section, Figure 6 shows the necessary internal steps taken by ROS to enable the communication between two nodes, for illustrative purposes, the `dummy_sensor` and the `safety_node` were used.

Services

Although the topic-based publish/subscribe models are a flexible communication paradigm, its broadcast routing scheme is not appropriate for synchronous communications, which are often useful to achieve some design simplifications.

ROS provides Services as basic request-response pattern through a Remote Procedure Calls (*RPC*) implementation. This is a bi-directional synchronous form of communication. It's the preferable method when facing simple request/response synchronous interactions, avoiding the existence of decoupled processes.

Despite not being the ideal form of communication for the example presented during this chapter, the Listing 3.3 shows how the specification might be changed to use this communication model, assuming that the `dummy_sensor` has a `sensor_data` service. Being that, the new sensor might act as a server, and the new `safety_node` as the service client. The internal steps required to establish communication are similar to the ones taken in topic-based communications. The server node advertises its service when it's launched. After that, each service client must require the service URL to the master node, following with a standard RPC request to it.

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"

int vel;

void sensorCallback(std_msgs::Int64 msg){
    int value = msg.data;
    if (value > 10){
        vel = 5;
    }
    else{
        vel = 0;
    }
}

int main(int argc, char **argv){
    ros::init(argc,argv,"safety_node");
    ros::NodeHandler n;
    ros::Publisher pub = n.advertise<std_msgs::String>("safe_vel",10);
    ros::ServiceClient cl = n.serviceClient<std_msgs::Int64>("sensor_data");
    ros::Rate loop_rate(10);
    std_msgs::Int64 msg;

    std_msgs::Int64::Request req;
    std_msgs::Int64::Response resp;

    while(ros::ok()){
        if(cl.call(req,resp)){
            sensorCallback(resp);
        }
        msg.data = vel;
        pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

Listing 3.3: The new safety node implementation, as the service client, instead of using topics for communication with the sensor node.

From the presented C++ definitions, is obvious that the synchronous pattern of communication is not well-suited for this particular case. In fact, neither is for the majority of cases in the robotic systems, that usually operate under heterogeneous and non-fixed conditions.

Remember that, although the client/server were written both in C++, through the inclusion of the specific language proxies, the ROS *middleware* provides multilingual interfaces for the RPC or Topic uses. Thus, it's always possible to write both nodes in distinct languages.

3.1.3 *Launch Files*

A ROS application usually consists on a set of nodes cooperating between each others. Most robotic systems are multi-configurable, which may yield multiple software configurations as well. Thus, mechanisms that automatically deal with configuration issues that might emerge from this are of special relevancy.

There are various forms to deploy a ROS application, however, the most common is through the use of the *roslaunch*[3] command-line tool. This was designed to launch both, node collections as entire applications with pre-defined configurations. The alternative is to initialize and configure every node and parameter individually, which can easily became quite demanding and prone to human errors.

Thus, *roslaunch* uses launch files to perform initialization, rather than nodes. These are XML files, where the set of nodes that must be launched are described along with parameters and topic remapping. In order to illustrate the utility and specification process of launch files, the Figure 7 shows two distinct configurations for the same robotic application. The nodes specifications are based on the ones that were previously presented during the topics section 3.1.2. As said before, the `safety_node` receives data from a sensor, and sends velocity commands through a `safe_vel`, which will be used by an unknown ROS component to move a given wheel base. For illustrative purposes, let's consider the existence of two distinct mutually exclusive sensors. Both based on publish/subscribe models of communication.

Each configuration can be easily managed and automatically launched through the simple execution of a single file. The specification of each file is shown on Listing 3.4 and Listing 3.5, respectively. Notice that, although this configuration files are pretty straightforward, in more complex systems they might be used to define parameters in the *parameter server*, arguments passed by command-line, or even to launch more complex groups of nodes under some specific circumstances.

3 https://wiki.ros.org/roslaunch

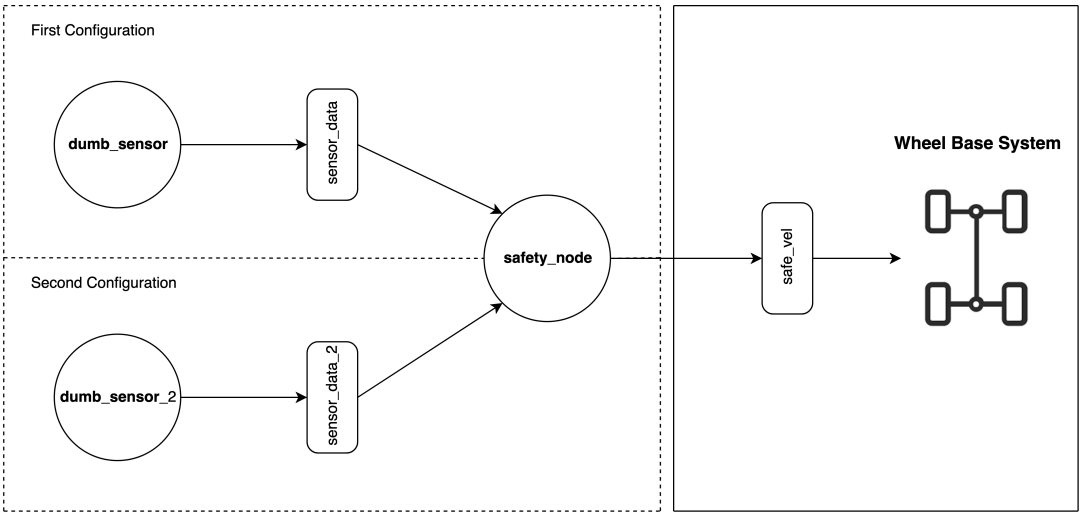


Figure 7: The Figure depicts two possible configurations of the same robotic system. Dashed lines are used to illustrate mutually exclusive components. The first configuration shows how the first `dummy_sensor` and `safety_node` are logically connected through the `sensor_data` topic. The second one shows a slightly distinct configuration, where a different sensor is used.

```
1 <launch>
2     <node name="dummy_sensor" pkg="example" type="active"/>
3     <node name="safety_node" pkg="example" type="active"/>
4     <include file="$(find example)/launch/wheel_base.launch" />
5 </launch>
```

Listing 3.4: The launch file that can be used to launch the first configuration of the sample robotic application.

```
1 <launch>
2     <node name="dummy_sensor2" pkg="example" type="active"/>
3     <node name="safety_node2" pkg="example" type="active"/>
4     <remap from="sensor_data" to="sensor_data_2"/>
5     <include file="$(find example)/launch/wheel_base.launch"/>
6 </launch>
```

Listing 3.5: The launch file that can be used to launch the second configuration of the sample robotic application. The remap tag transparently re-route the name supplied to the primitive. It allows to connect the alternative sensor without altering the `safety_node` source code.

3.2 QUALITY ASSURANCE

As the robotic software is being increasingly used on wide domains, namely in critical scenarios with human interaction, the quality assurance of these systems is becoming a matter of special interest. In such cases, there is a high required flexibility and the consequences of robot malfunctions due to software errors may be unacceptable. Therefore, it's imperative the use of tools and software engineer techniques that promote the system's quality. This section will introduce some state-of-art tools and approaches focused on the analysis of robotic software and variability issues. Although this chapter was dedicated to the concrete exploration of ROS applications, with regard to the QA matter, there isn't a clear borderline detaching the ROS challenges from other common multi-configurable robotic software. Thus, besides QA approaches taken under ROS systems, during this section, a more wide spectrum of systems might be considered.

3.3 STATIC ANALYSIS

Through the design and development process it is imperative to have tools that provide simple test and verification procedures. More specifically, it is necessary to have ways to observe a system state at a given time, during its execution. The *rqt_graph*[4] is one of the tools in which the developers most rely on. It provides a visualization of the system computation graph with run-time statistics. However, in order to perform analysis on critical and complex environments, this kind of run-time analysis is not convenient as trustworthy. Thus, better strategies to perform analysis at compile time are required. Although research and tools on static analysis of ROS systems are scarce, there are few which are relevant within the context of this dissertation.

Most of the works on QA of robotic systems are usually focused on the analysis of very narrow specific points. For instance, a proposed technique [24], backed by the *PhrikyUnits* [25], was introduced as an automatic static analysis tool to detect dimensional inconsistencies in ROS C++ code and its messages, relying on the prior annotation of the standard libraries.The incorrect use of the standard data structures that represent physical quantities can't be detected by the compiler, this leads to possible system faults at execution time. Through the use of this tool, a well-succeeded study on the frequency of dimensional inconsistencies was performed in [26]. The tool was used to perform analysis upon 5.9M lines of source code from sample repositories. Despite the positive results, this work was limited to the characterization of the physical units usage, and to give an insight on how its manipulations might promote software faults. Thereafter, this work has also been extended to support the probabilistic inference of unitis from non-annotated libraries [13].

4 http://wiki.ros.org/rqt_graph

Some approaches extract models from the code. In [27], was proposed a technique that statically extracts C++ ROS code, creating a model of the message flow between components, which is then used to highlight components that are affected by code changes. The work presented in [34] analyzes the impact on a system when a specific publication rate is changed. The impact of rates on overwrite buffers, that might cause overflow and message loss is a major factor to have in consideration when analysing ROS architectures. The work has described a methodology with great potential to reduce the size of the expected impact set to half. The contribution value was on the capacity to reduce the needs to re-apply analysis under applications that have suffer slightly changes.

Beside focus-oriented static analysis tools, has the ones presented above, the work performed on broader analysis of ROS applications is almost nonexistent. The more relevant contribution was initially presented in [30]. *HAROS* is a plugin-driven framework to perform an automatic static evaluation of the ROS code quality. Its main focus is to assist the development process by employing a set of diverse analysis techniques, reporting to the developer, quality metrics as violations to code standards. The considered code metrics vary from the simple Number of Code Lines (*LOC*) to the method Cyclomatic Complexity (*CC*). Since the framework is plugin-driven, further develops were made and included in on it. In [32], with the aim to enhance static-time support, a metamodel to describe ROS application architectures was proposed. In the same work a code query language is also provided. This is used to detect simple architectural patterns. This work promotes the comprehension and analysis of ROS applications, and encourages its use in more sophisticated techniques that can be developed as HAROS plug-ins. One of such examples was its use as the basis of a ROS property-based technique [31][33].

Although the HAROS framework, its plugins and language represent major contributions to assess the improvement of the ROS software quality, there are room enough to explore and develop new approaches as framework plug-ins. Thus, taking advantage, and going beyond the work that already has been done, new approaches to increase the software quality might be explored. Namely, a possible value approach might be the integration of formal methods to interact with the framework data.

3.4 PROPERTY VERIFICATION

Towards properties verification, there has some work been done, namely some attempts to verify safety properties of ROS applications through state-of-art model checkers. Into this context, some approaches were taken, namely using SPIN [37] and UPPAAL [11]. However, these are only exploratory approaches based on ad hoc codifications of robotic software into the target model-checking language.

Outside the ROS orbit, there are some works on robotic property verification that deserve to be mentioned. In [18] an end-to-end dependability case for the control software of a family of surgical robots, was constructed. A safety-critical property was chosen and a case was built through a pipeline based on lightweight formal analysis (Alloy), feature modelling and testing. Even though the positive results, the work stills has presented as the major limitation the requirement of verification expert team.

Into this context, another case study report was presented in [22], the novelty in this work was the methodology approach based on a trade-off between ambitious context-insensitive analysis and user-intervention analysis based on domain-knowledge providing.

Despite the positive results in some of these works, they're not convenient or easy to apply it in the majority of cases. The ROS software community as its appliance domain are growing, and its proximity and knowledge with regards to formal methods is scarce. Thus, it seems that the more convenient approach to the property verification and formal analysis of ROS software is by its integration through automatic procedures.

4

VERIFICATION OF ROS SYSTEM-WIDE SAFETY PROPERTIES

The robotics software are flooding the industry across every possible domain of action. The possibility to increase productivity while decreasing costs is very attractive. Consequently, the responsibility delivered to automatic systems grows. The closer proximity between humans and robots, as well as the appearance of non-supervised robots, creates a vital necessity to safety certification.

Nevertheless, mainly due to the heterogeneous nature and the wide domains of action, large scale robotic systems are difficult to test and to analyse. Despite the attempts, the complexity of the actual systems made the integration of classic QA and verification techniques infeasible. Although there are some state-of-art frameworks and techniques to attest QA of such systems, they may be inserted into one of the two main categories:

- The first category, contemplates the mandatory techniques on the industry, which are based on software testing. Although, traditional testing techniques might be not appropriated to guarantee safety in some environments, such as critical ones, there are a few of more sophisticated ones emerging.
- The second category groups the more formal and less usable techniques. Here, the common issue is the inherit cost of usability over quality assurance. The use of such techniques requires high know-how, and therefore, are not accessible to the conventional developer of ROS software. Thus, its usage and portability might be confined to a very strict audience.

This work will try to fill in the gaps of each category by creating a bridge between less-formal and more-formal approaches. It intends to bring a solution that, can be used by the common developer, while resorting to formal techniques that may empower the state-of-art methods, towards safety verification. For being one of the most influential middlewares for robotic software, the analysis approach will be focused on ROS software.

ROS applications might be developed in multiple programming languages and be organized within heterogeneous architectures. When attesting the quality of modular systems such these, it is common to split the analysis into two stages:

- The analysis and verification of individual nodes behaviour. From Unit testing, up to verification procedures. Depending on the software construction, the validation of the relations between input and output are usually possible to be confirmed through standard techniques.

- The analysis and verification of the system behaviour. Here, end-to-end properties are analysed assuming the correctness of the individual components. The main issue when performing analysis at system level, is the massive cost of launching an entire system. Which in these cases, are usually unsustainable. Therefore, the usual testing techniques used to perform system analysis are unable to provide acceptable levels of guarantees.

During this work it will be assumed the correct nodes behaviour in order to enable the system-wide verification. By seeing individual nodes as black-boxes, and relying on its loose behaviour specifications, a novelty technique using Electrum to automatically verify system-wide safety properties for ROS applications will be presented during the following sections.

4.1 MODEL-CHECKING ROS SAFETY PROPERTIES

To perform formal verification of ROS applications in Electrum, it is necessary to arrange a precise definition of its concrete structure and behaviour. Every application makes use of distinct resources and has distinct behaviours. However, there is a set of logics and features that represent the fundamentals of every application, since they are build over a common ground.

Since, in Electrum there is no clear distinction between the model and its specification, performing analysis might require high-levels of decision making. Besides that, due to the language flexibility, the same model or specification can be achieved doing very distinctive choices without compromising results. Therefore, when predicting a possible process automation, it is required to have a precise and consistent definition of a possible specification.

Being that the majority of ROS applications have complex structures, developed within large scale environments, considering the hypothesis of practicing unbounded verification seems unrealistic. Thus, this work intends to demonstrate evidence to the possibility of automatically performing bounded-verification. The Electrum seems suitable to put this into practice. It is flexible enough, supporting both architectural as behavioural properties specification. Besides that, it's supported by a back-end engine that performs bounded-verification based on SAT solvers.

4.1.1 *ROS Meta-Model*

To analyse general ROS architectures, it's necessary to establish the common ground of each application starting by defining the fundamentals for the family of every possible ROS product. The problem may be reduced to the specification of the middleware structure and behavioural constraints. Mainly due to the nature of ROS applications, the first version of this work will focus only on the analysis of topic-based architectures. The Listing 4.1 shows a possible model specification of the middleware. In practice, this represents the meta-model of every possible ROS application.

Thus, the ROS meta-model specification consists essentially on:

- A set of top-level abstract signatures: `Node`, `Topic`, `Field` (lines 3 - 8). The first two signatures represent the abstract notion of a Node and Topic. The abstract label means that, there is no instances of those objects without explicitly extending them. Basically, it defines the general concept of a Source Code Entity in ROS. Their extensions will define more precisely the concept of a Named Source Entity, while its instances will represent the respective associated runtime object.

- A abstract `Value` signature (line 9). It depicts the abstract notion of Message data-type value. These data-types may reduced to a set of different primitive typed field values. Its extensions, the `numeric` and `string` (line 10) signatures are not abstract. Which means that, the notion of value will be split between disjoint sets of numeric or string primitive values. These are present at the meta-level since its existence is transverse to the model definition.

- The Message signature. The only signature in this model that is not labeled as an abstract or extended from it. This is because a message is a global entity, whose definition is common to every possible application. A message is associated to a given topic, and has a set of Fields, each one with a given value. The relations within the signature declaration are well-defined by explicit declaration of their multiplicities and relation facts.

- One explicit fact with multiple declarations. The first declaration states that, in the initial state, the `inbox` and `outbox` variable relations should be empty. The remaining declarations are used to characterize the message flow within the model instances. These are very straightforward statements that constraint the system behaviour, namely by defining the volatility bounders of the variable relations. In the third and fourth declarations (line 25, 29) it is declared that every message in a given `Node` outbox should be eventually sent. The last declaration (line 32) prohibit the spontaneous appearance of messages in every `Node` inbox.

```
module rosmodel

abstract sig Node {
    subscribes, advertises: set Topic,
    var inbox, outbox : set Message
}

abstract sig Topic, Field {}

abstract sig Value {}

sig numeric, string extends Value {}

sig Message {
    topic : one Topic,
    value : Field → lone Value
}{
  some value
}

fact Messages {
  no (outbox + inbox)
  all n : Node | always {
    n.inbox.topic in n.subscribes
    n.outbox.topic in n.advertises
  }
  all m : Message | always {
    m in Node.outbox implies (all n : subscribes.(m.topic) |
      eventually (m in n.inbox))
  }
  always {
    all m : Node.outbox | eventually m not in Node.outbox
  }
  all m : Message | always{
    m in Node.inbox implies (some n : advertises.(m.topic) |
      before once (m in n.outbox))
  }
}
```

Listing 4.1: Common structure and behaviour specification for every ROS application. The top-level signatures depict the middleware resources. The fact describe the system constraints, both at the structural, as behavioural level.

Notice that, this meta-model is already architecture oriented, being that every relation within signatures define interface level concepts. Since the `inbox` and `outbox` relations are the only ones labeled as variables, the inner focus associated with the inter-node monitoring of message-flow, is directly captured.

4.1.2 *Systems Architecture based on Topics*

The model structure of each ROS application is divided into a structural and behavioural part. The structure depicts the system architecture, and defines precisely which nodes and topics are used, as well as the links between them. In practice, this will be reduced to the concrete instantiation of the abstract elements declared presented during the last section. Since the analysis will focus topic oriented applications, only that type of communications is considered.

A ROS application consists on a group of Named Source Entities precisely arranged. The Entities that are relevant to this analysis are the application Nodes and Topics. As well as the links between them. Each application might have distinct configurations, each configuration yields a distinct Electrum model. Through launch file inspection, the information regarding which nodes and topics are relevant to defining the structure are easily detected.

Besides the Nodes and Topics List of an application, there are some features that are required to provide the basis for a proper analysis. One of those is the messages data-types, which are nothing more than a set of `Fields`. Each topic has a specific data-type. In this structure, the relation between the topic and its data-type will be abstractly induced by an implicit relation between the topic itself, and a concrete set of Fields. This will be introduced in the structure through Electrum axioms.

The Figure 8 illustrates the relation between the meta-model, and each concrete model regarding a concrete application architecture. By extending the interface provided trough the meta-model, each relevant feature is defined as a concrete signature, or implicitly declared through axioms. Besides the relation between Topics and Fields, these axioms will define the static relations `subscribes` and `advertises`, spelling out the concrete links between the both main structural signatures. Notice that, the figure only depicts structural concepts, leaving out the representation of behavioural properties. This will be made clear in the further sections.

Values Abstraction

The `Value` is a central piece of the analysis. Usually, the majority of safety properties intended to be verified regarding run-time behaviours. In such cases, relations between message values are the core part of the analysis. Safety system-wide properties in ROS are usually expressed as temporal relations between input and output message values.

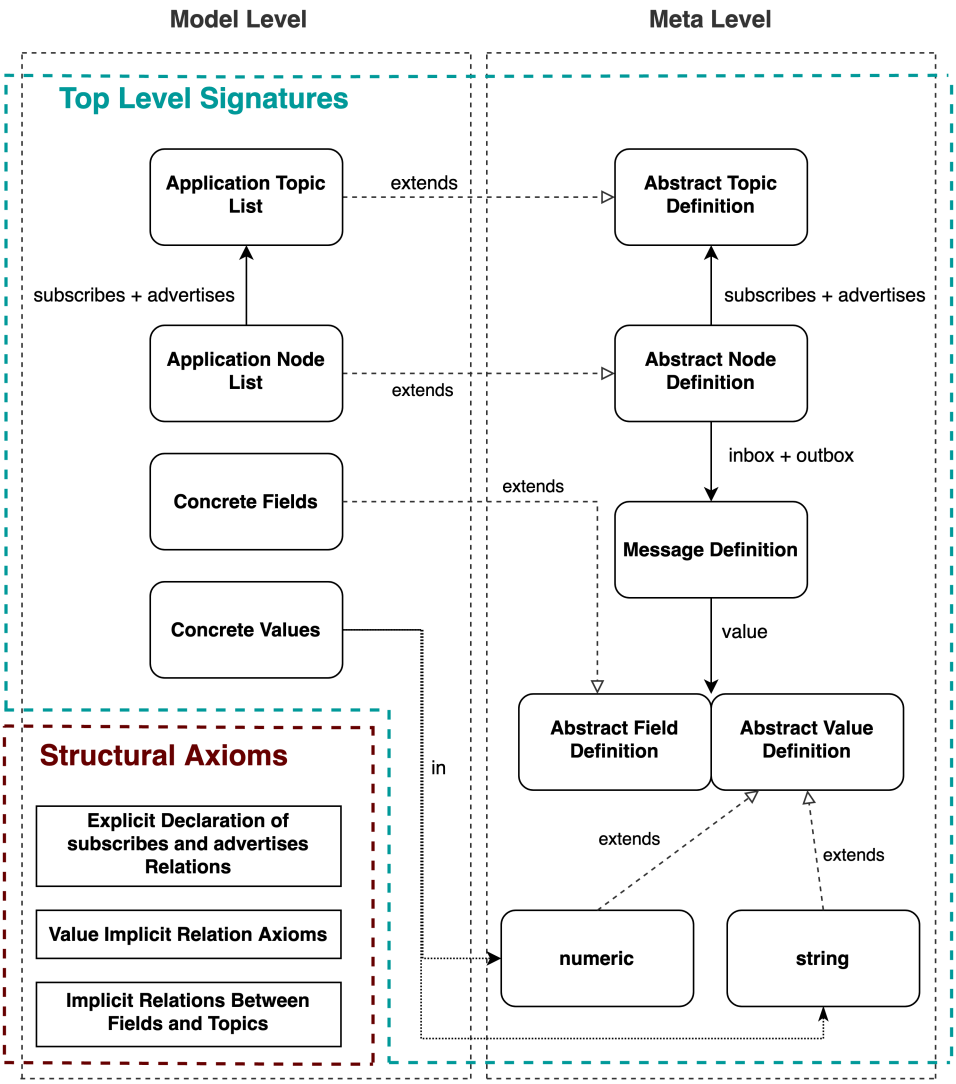


Figure 8: Representation of the main components within every model specification structure. The figure is divided through an two dimensional vector. Respectively depicting the distinction between the meta features from the middleware, and a concrete application specification. The second division represents the distinction between which features are captured through Electrum signatures and Electrum axioms. Solid connectors illustrate static or variable relations between top-level signatures, while dashed ones represent the hierarchical relations.

Considering that in ROS, every data-type is a more or less complex aggregate of primitive types, they all can be aggregated into two main categories, `string` or `numerical`. Dealing with `string` values represents no problem, its abstraction is made straightforward since the relevant properties are confined to equality relations. However, representing abstract numerical values while maintaining its useful properties is a non-trivial quest.

The bounded model-checking in Electrum, being SAT-based, is not suitable to deal with numerical values. Although the language supports the use of integers, these are not expressive enough to depict every possible value within the numerical types. Besides that, using integers will cause major impacts in scopes, turning the verification infeasible.

By taking advantage of the signatures flexibility and its hierarchy, the discretization of numerical values is made possible by interpreting them as intervals. The majority of system-wide properties in robotic systems are interested in reasoning about real values. These values are always discretized through the sensors. Thus, this approach follows the same guide of principles, creating a symbioses between the nature of the system reasoning and the respective specification. The approach introduces one main limitation, the absolute order between numerical values is lost, as well as the capacity to produce arithmetic operations. However, it enables to practice verification in slightly constrained scenarios, where inclusion and equality are the only relevant relations to the properties specification.

Figure 9 shows the `Value` signature hierarchy. The Division of the figure in layers relate the different levels of abstraction required to interpret concrete values. The First layer defines the meta-model abstract `Value` declaration. The second one captures their extensions, which represent disjoint sets, namely the `string` and `numerical` ones. The third and last layer is where concrete values are captured. This occurs after a possible behaviour be specified. Since before that, it's impossible to unravel which values are relevant to the analysis.

In Electrum, a scope limits the number of atoms that are allowed to co-exist within the same signature set. Thus, each concrete value will be inhabit for a set of atoms, that in some cases, may be singletons. Mainly in `numerical` cases, the intervals are implicitly related with each other, through full inclusion and full independence axioms. When nothing is said about them, the minimum set of constraints are those implicitly stated by the hierarchy signature. That is, different intervals which don't maintain a full inclusion or full independence relation, might have atoms that are free to navigate between them.

4.1.3 *Systems Behaviour*

The ROS meta-model provides a structural interface for specifying every ROS application structure. However, nothing has been stated regard the specification of both, nodes behaviours and properties. These are a crucial part of the analysis since most safety properties are temporal assertions about system-wide behaviours. In Electrum, due to the language

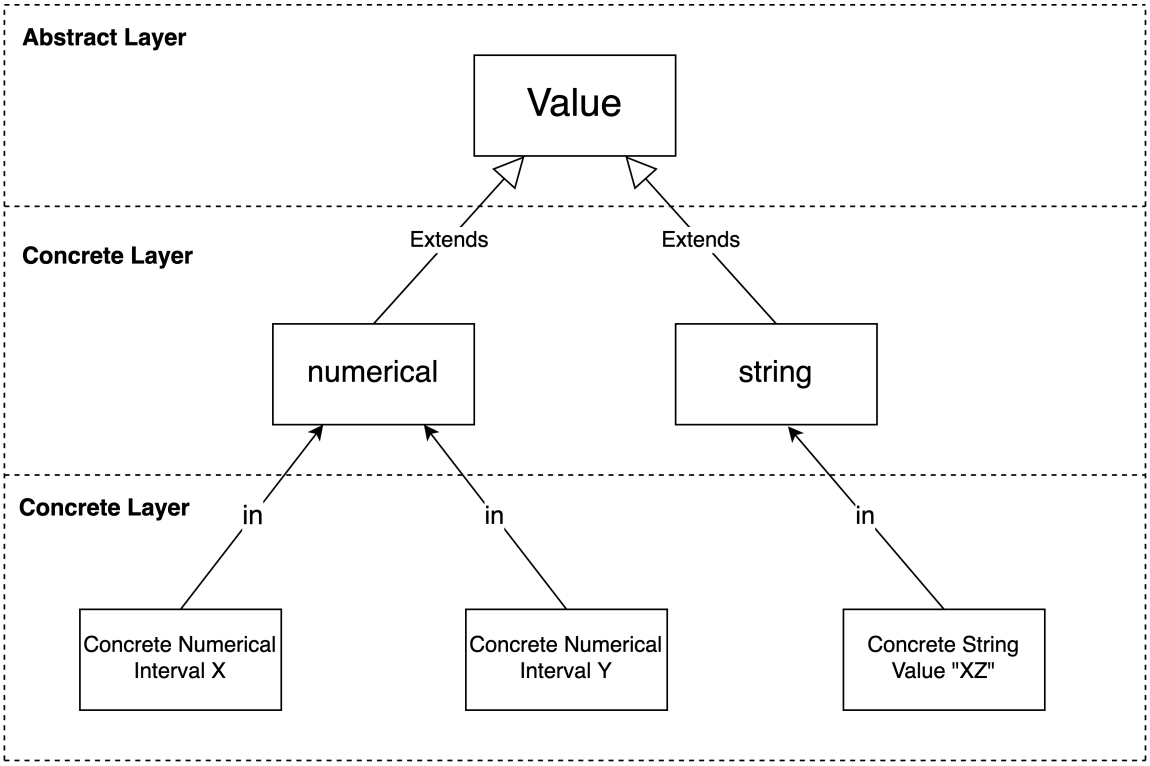


Figure 9: The three layers illustrate the distinct levels of abstraction on the value discretization. The first one is an abstract layer, defines an interface through at the meta-model level. The second is defined through extension, defining two disjoint sets of distinct value categories. The third complements the second through including specific values into one of the two categories.

flexibility, it is possible to design novel idioms as patterns to express abstract state transitions. Since the main goal of this work was focus on achieving an automatic form of property verification, there was a special need to design a new idiom based on generic specification patterns. The Electrum temporal-idiom to express systems behaviours was conceived though the following guide lines:

- The pseudo-formalization of the idiom should be driven by the Dwyer's [8] specification patterns. These patters were identified for the community as the the most typical. As will be further explained, this will be a central piece on the systems behaviours deduction.
- Being that there is an intention to automatize the process, a loss of information during possible translations are not critical. However, the incorporation of additional information might be problematic by affecting the soundness of the verification processes. Thus, the idiom must be able to capture the essential information regarding the systems behaviour.
- As there is no clear distinction between the model and the specification in Electrum, both properties and node behaviours should be written through the same idiom. This eases the specification process, since the behavioural specifications will be globally made at an interface level. This will support decisions that will be made clear during further sections.

The Dwyer's patterns are a standard form of specification for finite-state verification. These are a collection of patterns that commonly occur in the specification of concurrent and reactive systems, as ROS applications. Although the idiom was not projected to embrace every possible pattern, there are two groups of special relevancy that were contemplated. The first address the specification of safety properties. These can be separated in two subgroups, the ones that can be viewed as pure safety properties, and the other ones based on conditional events. The first is known as the *Absence* pattern, while the second is named *Precedence*. It follows a subtle explanation based on a concrete example, which has been previous presented during the Section 3:

- The *Absence* pattern is used to forbid undesirable events. One might state that, "Every message that goes through the `sensor_data` topic has data values between zero and one-hundred". Or the equivalent, through its negated form ("There are no messages passing in the `sensor_data` topic, with a positive data value without being between zero and one-hundred").
- The *Precedence* pattern allows the occurrence of an event when other required event has happen before. For instance, "If a message with a data value equals to zero passes

through the topic `safe_vel`, it's required that, previously, a message with data value between zero and ten has passed through the `sensor_data` topic.".

The second group of patterns regard the specification of liveness properties, and might be seen as its safety-duals. Although the aim of this work is the verification of safety properties only, these can be of useful hand when specifying individual nodes behaviours. Thus, the *Existence* pattern is associated to an event that shall eventually happen, at least once. While the second pattern is named *Response*, and is used to state that, after a trigger event happens, a response event will eventually happen. Following the above illustration based on the concrete example, it follows the exemplification of such patterns use:

- The *Existence* pattern is used to express inevitability. One might want to state that, "Some message with a data value equals to zero or five, will eventually pass through the `safe_vel` topic".
- The *Response* pattern is used to express a conditional inevitability. For instance, a property that states that, "If a message with a data value of one passes through the `sensor_data` topic, eventually, a message with a zero value will pass through the `safe_vel` topic.

Notice that, a property that may make sense on a given state, might be completely vacuous on another. Thus, the specification patterns consider the existence of temporal *scopes*, which essentially define the interval for witch the inner property must hold. Considering that ROS systems are inherited based on real-time events, this scope might be *globally*, delimited by specific events, or marked by real-time intervals. In this work, only globally scopes will be consider. Since the consideration of real-time intervals in abstract models goes way beyond this work context.

Assuming this limitation, the patterns descriptions are obtained straightforward in Electrum, by resorting to its temporal features. The specification of Absence and Existence patterns are achieved by transcription of simple statements. While Precedence and Response patterns, as conditional forms, are divided in two distinct types at the interface level. The first one, relates, through predicates, multiple precedence/response values of events. While the second, doesn't need to explicitly relate them. In Electrum, this is captured by simply using different quantifiers. Figure 10 illustrates how these patterns are interpreted on the Electrum temporal idiom. Notice that, as said before, the considered properties will be always under a global scope.

To obtain a better understanding on how these properties might be specified in Electrum, the Listing 4.2 shows a summarized definition of the properties that were stated during the Dwyer's patterns illustration.

This specification already presents discretized values, this was achieved following the procedure explained during the Section 4.1.2. The values inner-relations, as the relations

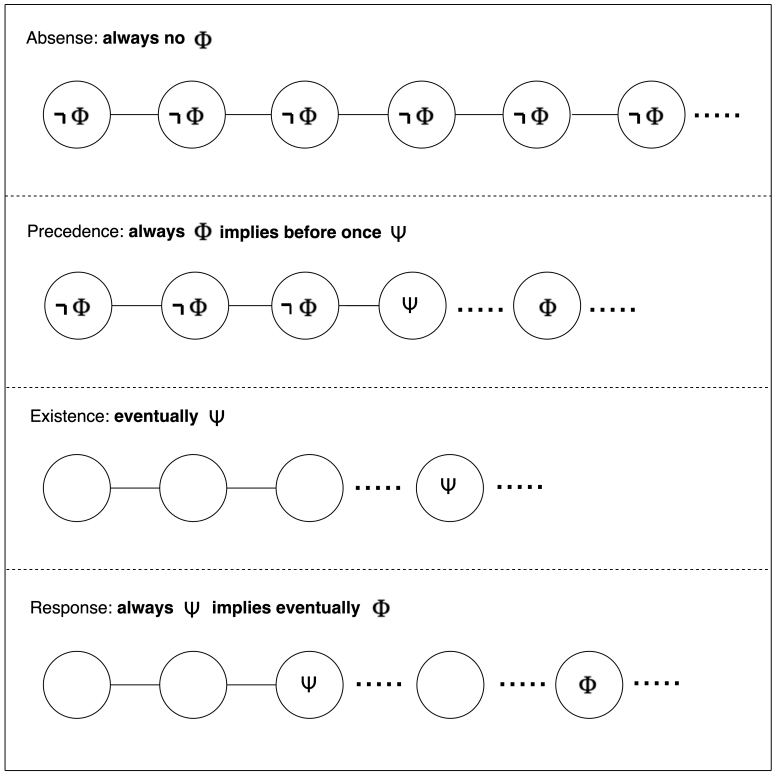


Figure 10: Electrum translations of the fourth specification patterns used by the Dwyer's approach. The Absence and Precendence patterns are used to specify safety properties. On the other hand, the Existence and Response patterns focus on liveness specification.

```
module properties

-- Fields Declaration
one sig data_sensor, data_safe_vel extends Field {}
-- Facts
....

-- Value Discretization
sig num_0 in numeric {}
sig num_1 in numeric {}
sig num_5 in numeric {}
sig range_from_0_to_5 in numeric {}
sig range_from_0_to_10 in numeric {}
sig range_from_0_to_100 in numeric {}
-- Facts
....

fact nodes_behaviour{
-- Absence Pattern :
  always{
    no m: Node.outbox & topic.sensor_data |
        data_sensor.(m.value) not in range_from_0_to_100
  }

-- Existence Pattern :
  eventually{
      some m : Node.outbox & topic.safe_vel |
           data_safe_vel.(m.value) in (num_0 + num_5)
  }

-- Response Pattern:
  always{
    (some m0:  Node.inbox & topic.sensor_data | data_sensor.(m0.value) = num_1)
      implies eventually
          (some m1: Node.outbox & topic.safe_vel |
              data_safe_vel.(m1.value) = num_0)
  }
-- Precedence Pattern:
  always{
    (some m1: Node.outbox & topic.safe_vel | data_safe_vel.(m1.value) = num_0)
      implies before once
          (some m0: Node.inbox & topic.sensor_data |
              data_sensor.(m0.value) in range_from_0_to_10)
  }
}
```

Listing 4.2: Property specifications following the Electrum temporal idiom. Each property translates the respective definition that was presented in textual form during the illustration of the distinct Dwyer's patterns. The signatures declarations are used to illustrate how the values and fields abstraction its using when describing behavioural facts.

between fields, values and topics, were all hidden for the sake of readability. These will be presented during further sections.

4.2 HAROS INTEGRATION

The HAROS framework, which as already been briefly presented, is a genericc plug-in-driven QA framework for ROS. It allows the evaluation of code and systems quality, mainly through static analysis procedures. Aiming the automation of the verification process, an integration with the HAROS eco-system provides both the required resources to the technique appliance as well as a well-stablish infra-structure, providing all the facilities for the technique integration and its user-interface.

Latest HAROS developed features allow the automatic extraction of the system structure and the specialists behavioural specifications. The first is an embedded feature that provides a structural meta-model, while the second is a semi-integrated Property-based testing approach and the respective specification language.

During this section, these features will be further explored, presenting both its value and as utility, within this work context. Besides that, will be given a complete overview of a new HAROS plug-in, designed to perform automatic verification of system-wide safety properties.

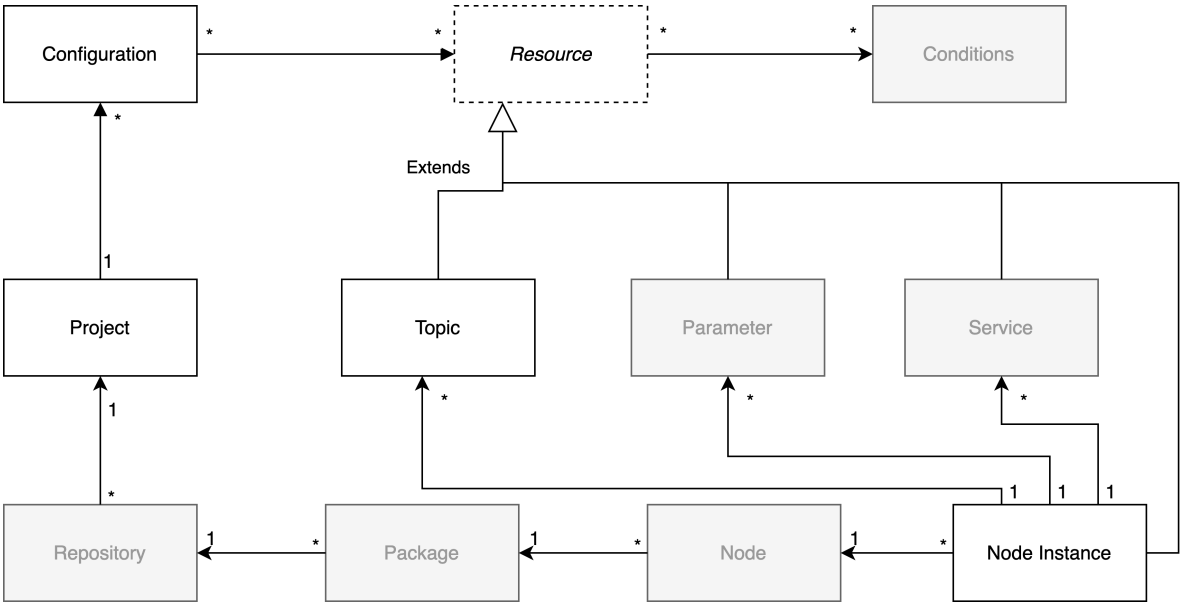


Figure 11: The HAROS architectural meta-model, the features that are unsupported by the verification technique are greyed out.

4.2.1 *HAROS Architectural Meta-Model*

The HAROS framework provides two main plug-in entry points for information relating ROS applications under analysis. One of them is devoted to packages and source files analysis, the other is for architectural information depicted through meta-models based on launch configurations. These models are extracted and generated automatically through source code analysis.

The verification technique proposed during this work will use these meta-models to capture the structural description of each application. By containing information regarding the launched nodes, and which topics are subscribed and advertised at run-time, each application meta-model provides the required data to automatically build its corresponding structural specification in Electrum. Besides nodes and topics, there are other computation graph elements, as services or parameters, that will not be supported by this technique first approach. The architecture of the HAROS meta-model is illustrated in the Figure 11, where the relevant elements to this approach were highlighted.

The direct correlation between the HAROS meta-model and ROS computation graph entities its very clear. This was a design choice that was guided the implementation of the verification approach, since it promotes the readability and result interpretation by a conventional ROS user.

```
property    ::= scope : pattern
scope       ::= globally
              | <|after activator [until terminator]|>
              | <|time-bound after activator|>
pattern     ::= some events | no events
              | events causes <|[time-bound]|> events
              | events requires <|[time-bound]|> events
events      ::= multi-event (|| multi-event)*
multi-event ::= event | <|event-set|> | <|event-chain|>
event       ::= name [predicate] [as ident]
predicate   ::= { condition (, condition)* }
condition   ::= param-expr binop value-expr
param-expr  ::= param | <|builtin|> ( <|param|> )
value-expr  ::= set | range | value
set         ::= [ value (, value)* ]
range       ::= number to number
value       ::= number | string | ref-expr
ref-expr    ::= [$ident .] param | <|builtin|> ( [$<|ident|> .] <|param|> )
binop       ::= = | != | in | not in |<| < |>|<| <= |>|<| > |>|<| >= |>
param       ::= field [. field]
field       ::= ident [[ (n-zero | <|slice|>) ]]
```

Figure 12: The HAROS behavioural language. The features that are unsupported by the verification technique are greyed out..

4.2.2 *HAROS Specification Language*

The HAROS framework provides a specification language that allows to express systems properties and its expected behaviours. From this language its possible to automatically generate runtime monitors and tests for Property-based Testing (PBT) practice.

Property-based Testing is an automated testing technique that, given a property, randomly generates test inputs, executes them, and verifies if the relation between the input and output respects it. Despite the required degree of formalism in conventional PBT frameworks, the specification in HAROS is made easier. This happens because there is a high-level of similarity between its specification language and system related concepts, which are familiar to any ROS developer. The core of the language is based on the Dwyer's specification patterns, that were already mentioned, during the Section 4.1.3.

This domain-specific language, whose syntax is shown in Figure 12, acts at a message-passing level, treating nodes as black-boxes. The semantics is formalized in metric FOLTL over ROS execution traces. Both the logic's type as the specification patterns create a direct match between the language and the Electrum temporal-idiom, developed during the course of this work.

Notice that, the HAROS testing approach is system-oriented. When varying the test scope from individual nodes to the whole system, the testing is done the same way. This happens

because the properties are always focusing interface functionalities, dismissing all the inner behaviours.

Despite the broader, and more efficient approach that was taken by the HAROS, when compared with standard testing techniques, there is a lot of opportunity to enhance its use through more sophisticated techniques. Namely by exploring new approaches to enhance the quality assurance through model-checking. Although HAROS has presented a specification language, this has been only used for testing. The physical limits that are imposed by the usually large robotic systems, tend to require high costs of deployment. Being that, as the system needs to be rebooted for each test-case, the cost of testing a test-suite with a reasonable size may be unbearable. Therefore, the coverage provided by such techniques leads to poor quality assurance regarding system-wide testing.

4.3 MODELLING THE DUMMY ROBOT

For illustrative purposes, let's recover the first configuration of the Dummy example, which was firstly introduced on the last chapter, and mentioned later on the last section when specifying temporal properties. In that case, four distinct properties with a system-wide flavour were written. Although the properties had a general purpose, that doesn't need to be always the case. In some concrete situations, narrow definitions referring individual nodes can be stated. For making sense of this example, let's assume that, each one of the four properties are an expected behaviour of some concrete node. The Listing 4.3 shows a possible definition of how these properties might be used in property-based testing through the HAROS specification language.

The specification is interpreted within a YAML file, where each node functional-properties are specified following the previous presented patterns. It follows with a description of the system-wide properties that must hold for each configuration.

The first properties (lines 12,19,21), are focused on specific node instance types. This means that, HAROS will monitor individual nodes for specific test-cases, which are generated through the properties written within each node label. The remaining properties, regard specific system configurations, being therefore, interpreted more widely as architectural test-cases. At verification level, there is an implicit interpretation, which bases the assertion of the system-wide behaviours when its assumed the correct operation of all individual nodes. Basically, the configuration properties are expected to hold when the set of node properties are considered axioms.

When merging all the possible information to be collected through the HAROS infrastructure, and the testing specifications, the elaboration of an Electrum model to perform bounded-verification on the configuration property becomes a straightforward process. The first step is to define the application structure, through the extension of the Electrum meta-

```
project: dummy_project

packages:
  - dummy_package
  - wheel_base_package

nodes:
  dummy_sensor_node:
    rosname: dummy_sensor
      hpl:
        properties:
          - globally: no /sensor_data {data not in 0 to 100}

  safety_node:
    rosname: safety_node
      hpl:
        properties:
         - globally: some /safe_vel {data in [0,5]}
         - globally: /safe_vel {data = 0} requires
           /sensor_data {data in 0 to 10}
         - globally: /sensor_data {data = 1} causes /safe_vel
           {data = 0}

configurations:
  first_configuration:
    launch:
      -dummy_package/first_configuration.launch
    hpl:
      properties:
         - globally: /safe_vel {data not in 0} requires /sensor_data
           {data not in 0 to 10}
```

Listing 4.3: A HAROS configuration file for property-based testing. Each set of properties will generate a specific test-suite. The nodes have property fields, these are used to generate test-cases to submit on concrete Node Instances. In the other hand, configuration properties will generate broader test-suites, whose monitoring will focus on an entire configuration. Each configuration is described through a concrete launch file.

```
module DummyRobot
open rosmodel

-- Topics Declaration
one sig sensor_data extends Topic{}
one sig safe_vel extends Topic{}

-- Nodes Declaration
one sig dummy_sensor extends Node{}{
    subscribes = none
    advertises = sensor_data
}
one sig safety_node extends Node{}{
    subscribes = sensor_data
    advertises = safe_vel
}
```

Listing 4.4: Parcial Electrum structural specification of the DummyRobot, following a structured form of specification.

model interface. The HAROS meta-model can be used to extract all these information. The Listing 4.4 presents the concrete specification of both Nodes as Topics for the Dummy Robot.

The second step is to define concrete values, concrete fields, and establish every required implicit relation between these entities and the remaining model entities. Each message has a concrete topic, which implicitly constraints which data-types are the valid ones. Since the notion of data-type has been dismissed from the signature hierarchy, an implicit relation associating each topic message with a valid set of fields must be written. These inter-signature relations are not unique, since a field needs always to be associated with a value category (string or numeric). Thus, both values, fields, and the respective required facts, are shown in Listing 4.5.

Notice that, for some values a `Singleton` axiom was declared. This helps the semi-automatic computation of the minimum scopes required to verify properties within the specification. Since intervals with one single element doesn't need to have more than one atom.

Finally, the model is completed through the specification of the node behavioural axioms, and configuration assertions. The nodes axioms are conceived through translation of each node property from the respective HAROS PBT specification. The assertion that is passible to be checked, regard the configuration property. The Listing 4.6 shows the Dummy Robot model conclusion, considering only the verification of the unique system-wide safety property.

A real HAROS specification file for the Dummy Robot should contain more axioms, which should depict all the system behaviour. However, this specification was not obtained by an automatic procedure, and was intended to be illustrative. Thus, for the sake of readability, the set of translated properties was confined.

```
1  -- Fields Declaration
2  one sig data_sensor, data_safe_vel extends Field {}
3  -- Facts
4  fact type_coherency{
5      ((topic.sensor_data).value).Value in data_sensor
6      ((topic.safe_vel).value).Value in data_safe_vel
7  }
8  fact field_types{
9      (data_sensor + data_safe_vel).(Message.value) in numeric
10 }
11
12 -- Value Discretization
13 sig num_0 in numeric {}
14 sig num_1 in numeric {}
15 sig num_5 in numeric {}
16 sig range_from_0_to_5 in numeric {}
17 sig range_from_0_to_10 in numeric {}
18 sig range_from_0_to_100 in numeric {}
19 -- Facts
20 fact Singleton{
21     lone num_0
22     lone num_1
23     lone num_5
24 }
25 fact Independence{
26     no num_0 & (num_1 + num_5)
27     no num_1 & (num_5)
28 }
29 fact Inclusion{
30     num_0 in range_from_0_to_5
31     num_1 in range_from_0_to_5
32     num_5 in range_from_0_to_5
33     range_from_0_to_5 in range_from_0_to_10
34     range_from_0_to_10 in range_from_0_to_100
35 }
```

Listing 4.5: The second part of the Dummy Robot model. The values used in the configuration file are captured to the Electrum model. The axioms constraint the implicit relations between values. The fields are also captured from the configuration file. Axioms explicitly relating each topic with specific fields are stated. As well as, relating fields values which one of its extensions.

```
fact dummy_sensor_behaviour{
  always{
    no m: dummy_sensor.outbox & topic.sensor_data |
        data_sensor.(m.value) not in range_from_0_to_100
  }
}

fact safety_node_behaviour{
  eventually{
      some m : safety_node.outbox & topic.safe_vel |
            data_safe_vel.(m.value) in (num_0 + num_5)
  }
  always{
    (some m0: safety_node.inbox & topic.sensor_data |
    data_sensor.(m0.value) = num_1)
      implies eventually
          (some m1: safety_node.outbox & topic.safe_vel |
              data_safe_vel.(m1.value) = num_0)
  }
  always{
    (some m1: safety_node.outbox & topic.safe_vel |
    data_safe_vel.(m1.value) = num_0)
      implies before once
          (some m0: safety_node.inbox & topic.sensor_data |
              data_sensor.(m0.value) in range_from_0_to_10)
  }
}

check first_configuration_prop0 {
    (some m1: Node.outbox & topic.safe_vel |
    data_safe_vel.(m1.value) not in num_0)
      implies before once
          (some m0: Node.inbox & topic.sensor_data |
              data_sensor.(m0.value) not in range_from_0_to_10)
} for 4 but exactly 3 Value, 3 Message, exactly 5 Time
```

Listing 4.6: Node behaviours and properties specification. Both based on the specifications extracted from the configuration file presented in the Listing 4.3

4.3.1 *Model-Checking Plug-In*

With the aim to develop a framework which applies formal techniques within less-formal environments. The approach focuses on the use of Electrum to specify and verify ROS system-wide safety properties. The technique is wrapped as a HAROS plug-in, using the resources that are made available through the eco-system, such as the UI or the file analysis features. The architectural HAROS meta-model is used as the main information entry-point. The remaining used information regards the specification files and a plug-in configuration file. Assuming the correct flow of information within the system, the plug-in implementation as follow the fundamental set of principles:

- The complexity of the translation procedures must be completly hidden from the user.
- Its installation and use must requires low levels of configuration.
- The plug-in must be able to capture both, structural and behavioural specifications automatically, through the HAROS infrastructure.
- The gap between the abstract and real models should be as little as possible. Thus, the analysis should be centred on reflecting only issues from the real system, discarding issues regarding the technique limitations.
- The plugin must produce readable results, based on concrete counter-examples. Which should be displayed through the HAROS user-interface.

The temporal-idiom enables the automatic creation of Electrum specifications. The specifications structure may be conceptually divided into four parts:

- Description of the ROS middleware.
- Description of a concrete application entities.
- Description of the system behaviour.
- Description of the verifiable assertions.

Each one of those parts represents a model refinement towards the final specification. The first is used to establishing the common structure and behavioural constraints, which in this case represents the middleware model. Follows the structure of a concrete application, where the named resource entities, as nodes and behaviours, are specified and explicitly linked. This mutable part of the specification may be automatically obtained through the HAROS meta-model entry-point. The `Value` discretization, the `Field` enumeration, and all its implicit constraints within the model, are obtained through the respective HAROS PBT specification, closing the model structural instantiation. Finally, the specification is also used

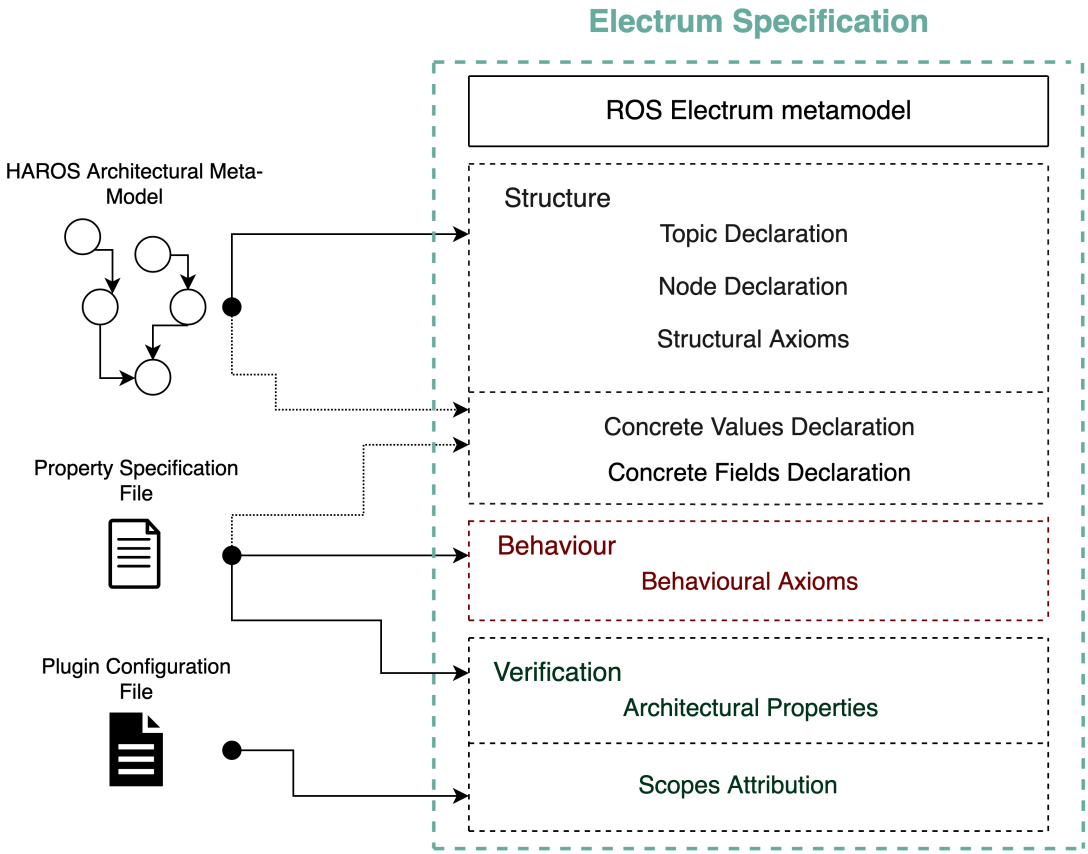


Figure 13: Conceptual parts of a ROS system specification in Electrum. The image shows the relation between resources and the specification blocks. The structure is obtained through the HAROS meta-model and the HAROS specification. The property scopes are introduced through a plug-in configuration file.

to describe each individual node behaviour through Electrum axioms, as the properties that must hold on each system configuration, as assertions. Multiple configurations of the same robotic-system are consider to be different models, since the HAROS eco-system already provides the necessary elements for abstracting this issue.

Figure 13 shows how this technique formalizes the launch configurations towards its possible verification through the Electrum Analyzer.

The main issues when abstracting real-time are usually associated to the characterization of concrete values. The approximation of Electrum specifications to the HAROS ones will always maintain a gap due to real versus abstract interpretations. Due to this, not every specification will be passible to be converted to this idiom. As said before, properties using time-bounds and scopes further than the **`globally`** one, will be completely discharged.

Although the first version of the plugin it only supports a confined set of the ROS systems characteristics, the approach might be easily extended to embodied new elements and fea-

tures. Breaking down the constraints, the approach is applicable to systems and specifications that respect the following requirements:

- The system must be topic-based. Although it is possible to analyze systems that contain synchronous patterns of communication, the specified behaviour and properties must be completely independent from those. This constraint is imposed at the top, by the non-existence of a good HAROS support for these patterns.
- A HAROS specification is prepared to translation, if its completely described through **`globally`** scoped specifications. Behavioural descriptions with other scope will not be accepted. This happens due to real-time issues, which goes beyond this work context.
- The HAROS specification should only contain single event-chains. Multi-event chains are discharged due to idiom and logic limitations.

The Figure 14 shows the main parts of the plug-in architecture. The user interacts with the plug-in through the HAROS interface. Being that the Electrum specification considers the node properties as axioms, the HAROS testing tool must be executed previously to the verification towards higher guarantees. After that, the plug-in automatically extracts all the required information to create an Electrum model for each configuration. During this phase, an intermediate structure, whose architecture is depicted in Figure 15, is created. The structure regards a specific configuration, creating and maintaining a link between the information from the meta-model, the behaviour specification, and the Electrum model thats generated from them. The model can be directly submitted to the Electrum Analyzer, where all of its assertions are verified for a given scope. If a given property is satisfiable by the model-checker, a counter-example depicting a time-trace is yield. The results are produced within a textual form, referring only model abstract concepts, that cannot be directly interpreted by the common user. Thus, by re-using the intermediate structure, the counter-examples are translated to its concrete-form, and retrieved to the HAROS interface with a event-based flavour, which might be very intuitive to the ROS developer.

After the creation of the intermediate structure, and before the Electrum model production, a set of optimizations are made. Since the extracted model is expected to contain a big chunk of innocuous information, due to the exportation of external ROS libraries, a possible optimization caused by the removal of this extra-data is certain. Thus, topics that are not ever used, or nodes whose functionally is useless at interface-level may be removed.

Every possible reduction in the number of the structure entities, causes a direct reduction in the number of minimal scopes, without interfering with the soundness of the verification process. During the experimental part of this work, it was verified a reduction of 50% in the `Node` scope, and a reduction of 70% in the `Topic` scope. Notice that, the experience is not representative, since was not largely reproduced. However, by empirical observation, seems a good direct approach to static scope reduction.

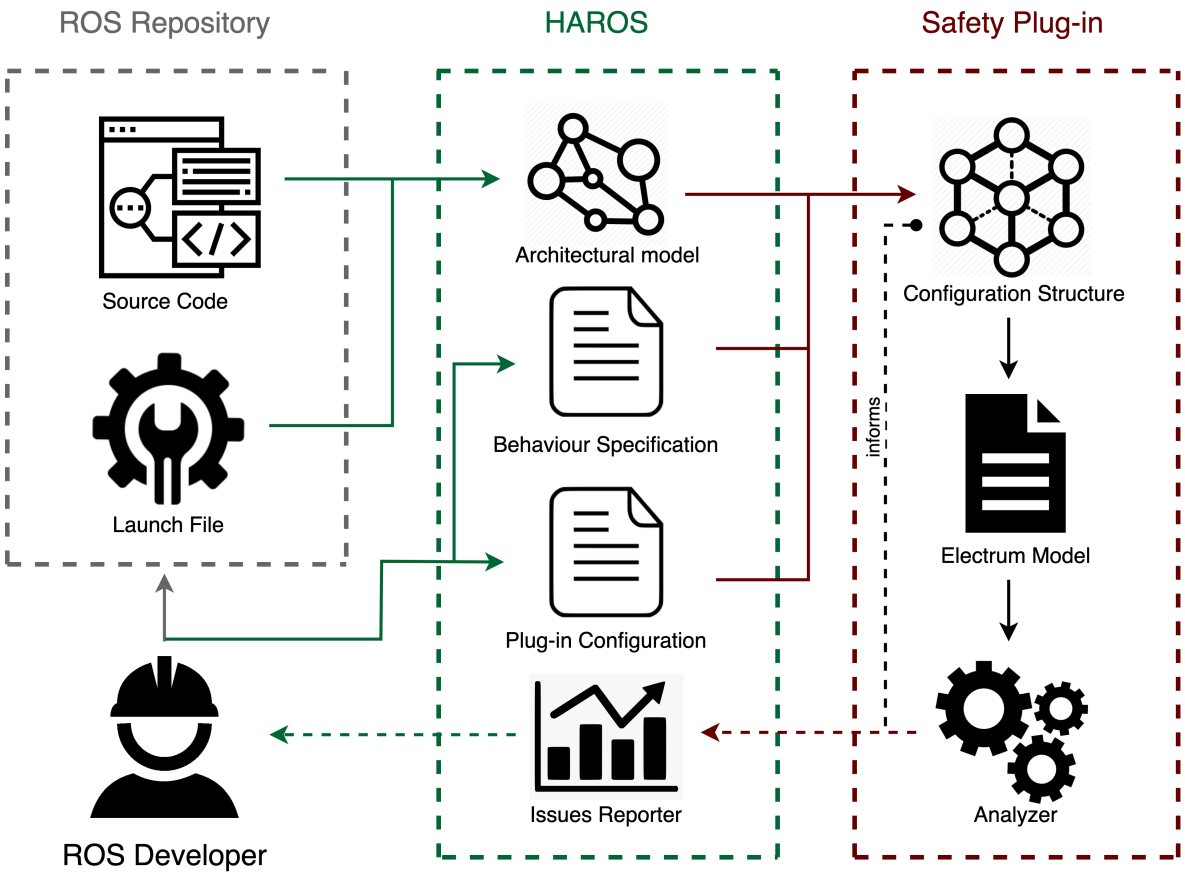


Figure 14: The Model-Checking plugin architecture main components. The plugin interfaces with HAROS through the infrastructure support features. The application outsources the model-checking techniques to the Electrum Analyzer. The results are retrieved through the HAROS user-interface. Solid arrows depict the flow of the information up to the plug-in, the dashed lines illustrates the results opposite flow.

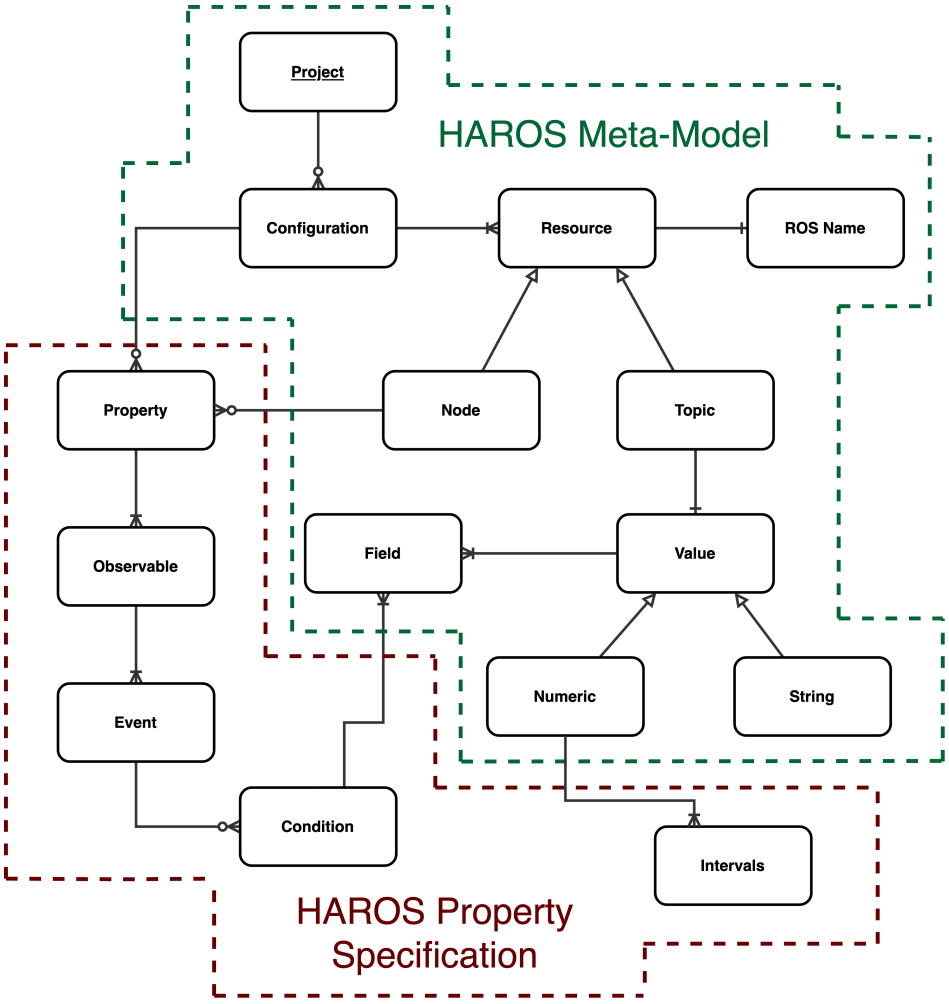


Figure 15: The configuration structure class diagram. Each class captures the information regarding the respective named source entities. The information is merged from two distinct highlighted sources, namely the HAROS meta-model and its properties specification.

The fact that every verification realized in this work is bounded, requires the use of scopes for each top-level signature. While `Node`, `Topic` and `Field` scopes are directly computed, `Values`, `Messages` and **`Time`** are not. This happens because different scopes of these signatures directly interfere with the range of behavioural possibilities. Despite being possible to compute an ideal approximation for these bounds, this could limit the plug-in usability, since it disregards the human intuitive capacity to perceive the system. An optimal approach is to have both, an option to explicitly specify the verification scope, while computing an optimal scope by default. The first version of this plug-in only contemplates the former. However, optimal scope deductions based on the number of distinct values, and the maximum radios of the computational graph, appears to be legit.

5

EVALUATION

This chapter presents an empirical evaluation on the use of the model-checking plug-in. The technique expressiveness and performance will be evaluated. Although not every HAROS PBT property is possible to be translated to the Electrum temporal idiom is expected that, the supported sub-set be enough to assess the safety on real systems.

Aiming to attest the technique expressiveness, a real industrial case will be analysed. This will be sufficient to unraveling both the capacity, as the viability of using this approach in real-contexts. Besides that, the analysis will provide enough results to evaluate the plug-in integration within the HAROS eco-system and its testing procedures.

Since the main challenges on bounded model-checking of real systems are usually conditioned by the scope sizes, the verification will be conducted for distinct bounds. This aims to provide a better comprehension of the approach limitations. Allowing a better partition of issues, that might be directly related to the approach expressiveness, or are simple inherit limitations of the model-checking techniques used.

Although the approach it should be able to support the verification of liveness properties, this was not the focus of this work. Thus, both analysis vectors will be focused on the verification of concrete system-wide safety-properties. All the evaluation will be executed using a SAT4J backend, on a 2.4 GHZ Intel Core i5 with 8GB memory running Ubuntu (16.04).

5.1 ROMOVI CASE STUDY

The RoMoVi [1] project's main objective is to develop robotic components in a modular and extensible mobile platform, which will provide commercial solutions for motorization and logistics of steep slope vineyards.

Developing robots for crop monitoring and harvesting is a complex challenge. The robot perception, localization, and mapping, must be precise. This may be specially difficult when operating on unstructured environments with low GPS signal as steep slop vineyards. In [7] is presented a platform intended for the measurement of variability in this kind of

1 http://agrob.inesctec.pt/

Figure 16: AgRob V16, the platform for the RoMoVi project.

environment. Among other innovative features, this robotic platform is equipped with an advanced navigation system that allows its fully operation when GPS signal is not fully available. Its development is being made in a modular way, being that, in a near future, besides variability measurements tasks, it will be used to work with robotic manipulators for pruning and harvesting operations.

The system aim is to be capable to act in unstructured environments while interacting with humans, providing quality assurances enough to its commercialization. Thus, methods to guarantee quality are absolutely required. The actual robot prototype (Figure 16) aims to accomplish a complete automatic navigation through the vineyards. This was developed incrementally, through the use of multiple configurations. There are a set of features and challenges that made this system suitable to attest the quality of the QA approach presented during this work:

- The AgRob platform is an industrial system, which may illustrate the challenges inherit to real case systems. Through this system it is possible to attest the applicability of this approach to real world robotics software.

- The system was designed and developed by a wide-range of professionals, which provide an unbiased sample of the typical ROS developer.
- The system is multi-configurable. Distinct configurations expect different properties to hold.
- The system relies on several third-party ROS packages, which makes it prone to suffer from launch configuration errors that may undermine safety properties.
- Being an autonomous system that will operate in a non-structural environments, non-expected system behaviours may have major consequences. Thus, attesting the system quality through less-conventional and more formal approaches, might provide higher quality guarantees.
- The system has a rich structure and a complex behaviour, being suitable to attest if the expressibility of the approach is enough to address the verification of relevant properties.

5.1.1 *Configurations*

The AgRob system has multiple configurations. Each configuration is described through a main launch file. Some launch files are hierarchically related, describing sets of distinct packages that should be grouped together. Through the RoMoVi formal documentation, four distinct configurations were identified. The analysis will focus only on the two main configurations, being that, all the analysis efforts on those may be applied to the remaining ones, expecting similar results.

Figure 17 shows a simplified AgRob architecture overview of both the main system configurations. Each configuration represents a different combination of features. The most basic configuration is named *startup*, and its launch file describes the minimum required set of resources to launch a robot, that is teleoperated through an external Joystick, while avoiding obstacles automatically.

The second configuration is named *map*, and diverges from the *startup* one by including localization and navigation packages. When on this configuration, the robot user can switch, from teleoperation to a completly autonomous mode. The first configuration only detects obstacles through laser sensors, while the second also considers localization information.

The core nodes of both are the `SafetyController` and the `Multiplexer` nodes, which are responsible for ensuring the system safety. The `SafetyController` monitors all the data coming from each sensor, as the execution mode, which is defined through the controller. Besides that, it constantly monitors the user operations, avoiding unintentional or even forced accidents. The robot current mode and status are always monitorized by this central piece,

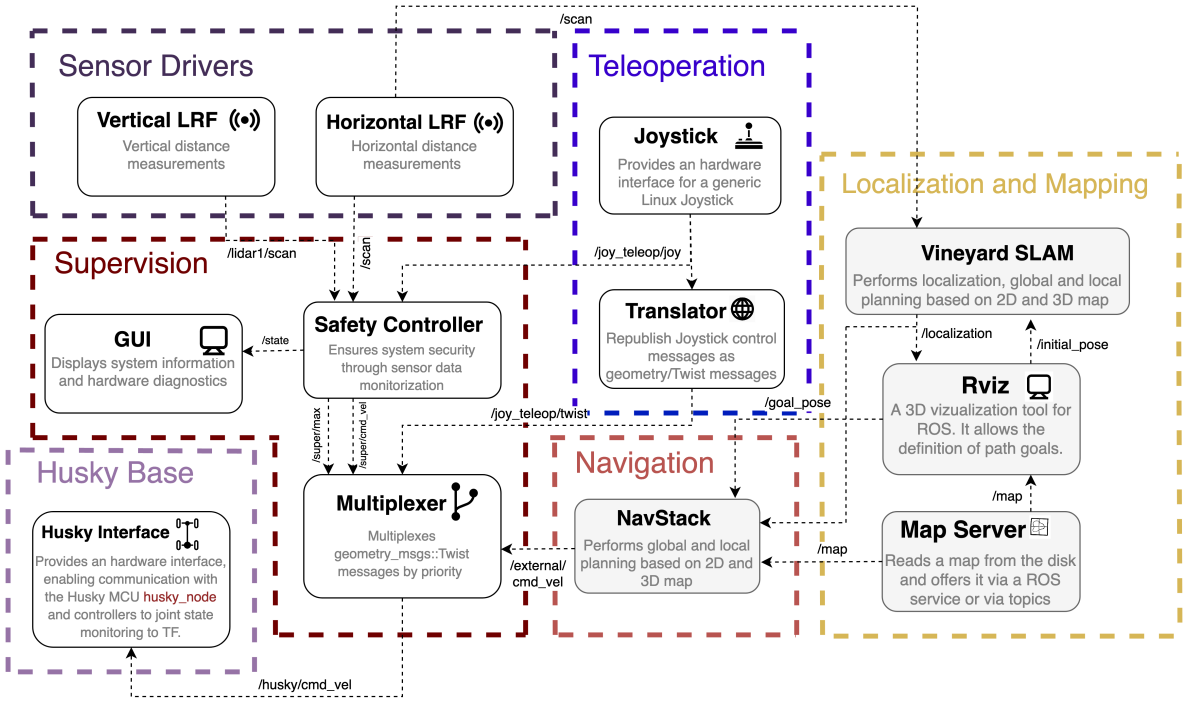


Figure 17: The AgRob V16 software architecture simplified. The components exclusive to the *map* configuration are grey out. The remaining components are present in both main configurations. Each conceptual division represents a given system functionality. In most cases, distinct functionalities result in different package groups.

which are constantly reporting feedback to the user GUI and safe velocity commands to the `Multiplexer` component. The `Multiplexer` collects commands from the different sources and multiplexes them following a pre-defined priority. The selected commands are then passed to the HuskyInterface, which communicates with the hardware actuators, making the robot base to move.

The map configuration differs from the startup by including a navigation stack, which is built upon a specific path planning, trajectory decision and computation external packages. These provides means to the system perform a proper form of simultaneous localization and mapping (*SLAM*)[21]. Besides that, it also integrates a 3D visualization tool to supervision and goal stipulation, enhancing the interaction possibilities with the user.

Notice that, in both configurations, as usual in most ROS applications, there is a big chunk of communications that occur implicitly through the tf[2] package. Although this package is built through well-tested standard libraries, a system-wide analysis should take them into account. However, there is no actual support for its analysis through the actual HAROS version. Thus, taking in consideration that the package functions are confined to maintain a coherency between robot coordinate frames, its analysis is not considered in during this first approach.

5.1.2 *Properties*

In most cases, system-wide properties consist on the specification of temporal relations between the system sensors and actuators message values, which are expected to hold within a given temporal scope. Notice that, every property passible to be verified through this approach needs to be globally scoped. Thus, the analysis will be confined to these properties.

Through the formal documentation and interaction with the system developers, a sub-set of system-wide properties that are expected to hold in both configurations were listed. These insights on expected behaviours regarding the values in one of both system actuators, namely the `Husky Interface` or the `GUI`. This happens because the message flow occurs from the sensors to the actuators. The `GUI` received messages contain a value in the field `data`, which is always associated with the actual `Joystick` operating mode. These are switched through combinations of binary values on the button fields. Messages with `button[0]` $= 1$ active the `Joystick` manual mode, if the `button[0] = 1` and `button[1] = 1` is detected, the robot will initiate a "go straight" mode. Besides that, when in joystick mode, `button[4]` and `button[5]` issue velocity commands. Taking this into consideration, interpreting the developer expected behaviours, the following properties are expected to hold for both configurations:

2 http://wiki.ros.org/tf

- Property 0: If the GUI receives a message with a `data` $= 3$, previously, a command with `button[0] = 0` and `button[1] = 1`, should have been sent through the joystick controller.
- Property 1: If the GUI receives a message with `data = 6`, previously, a command with a `button[0] = 1` should have been sent through the joystick controller.
- Property 2: If the GUI receives a message with `data = 0`, previously, a command with `button[0] = 0` and `button[1] = 0` should have been sent through the joystick controller.
- Property 3: If the husky base interface receives a message with a `linear.x` $= 0$ and an `angular.x` $\neq 0$, previously, or the joystick controller has sent a message with `button[0] = 1`, or a message with `range` smaller than 40cm ($0 \leq$ `range[0]` ≤ 4) has passed through the `scan` topic.

For validation purposes, it is useful to introduce properties that are expected to be false. Some properties are not expected to hold in both configurations. For instance, the following property is expected to hold in the `startup` configuration, and expected to produce a counter-example in the `map` one:

- Property 4: If the husky base interface receives a message with $1 \leq$ `linear.x` ≤ 10, previously, the joystick command should have sent a message with a `button[0] = 1` or `button[1] = 1`.

This is not expected to happen in the `map` configuration due to its navigation features. In this configuration it is possible to have a stipulated position goal, which the robot must reach without further indications through the `Joystick`.

5.1.3 *Specification*

In a first phase, every intra-node functional property must be specified and ideally tested through the HAROS PBT plug-in. Thereafter, each configuration property that is intended to be checked must be specified through the same configuration file.

The specification process is made interactively, the node specifications should follow the informal project documentation. When an adequate specification is achieved and every node has been successfully tested, it follows the specification of each system-wide property. Notice that, wrong node specifications can lead to false verification results. This detail compels a greater commitment when specifying and testing individual nodes, which results in higher-quality documentation and therefore, in high-levels of safety certifications. The final specification results on a complete definition of the nodes intra-functional properties,

Configuration	Spec	*n Value*, *n Message*, 10 *Time*						
		2	4	6	8	10	12	14
`startup`	Prop0	✓ (5.0s)	✓ (9.7s)	✓ (17.0s)	✓ (33.2s)	✓ (48.0s)	✓ (71.4s)	✓ (90.0s)
	Prop1	✓ (5.2s)	✓ (9.9s)	✓ (18.8s)	✓ (26.9s)	✓ (43.2s)	✓ (79.7s)	✓ (98.5s)
	Prop2	✓ (6.9s)	✓ (13.8s)	✓ (28.7s)	✓ (41.1s)	✓ (58,1s)	✓ (81.7s)	✓ (95.3s)
	Prop3	✓ (5.6s)	✓ (15.6s)	✓ (36.0s)	✓ (52.6s)	✓ (55.1s)	✓ (87.4s)	✓ (98.3s)
	Prop4	✓ (5.8s)	✗ (11.82s)	✗ (26.1s)	✗ (29.4s)	✗ (46.1s)	✗ (74.5s)	✗ (95.4s)
`map`	Prop0	✓ (5.1s)	✓ (10.2s)	✓ (17.7s)	✓ (36.1s)	✓ (60.2s)	✓ (79.4s)	✓ (98.9s)
	Prop1	✓ (6.4s)	✓ (11.6s)	✓ (21.8s)	✓ (32.8s)	✓ (44.8s)	✓ (66.2s)	✓ (96.4s)
	Prop2	✓ (5.2s)	✓ (11.1s)	✓ (21.4s)	✓ (32.4s)	✓ (51.0s)	✓ (68.8s)	✓ (111.7s)
	Prop3	✓ (5.2s)	✗ (9.7s)	✗ (16.4s)	✗ (27.8s)	✗ (38.9s)	✗ (51.5s)	✗ (84.4s)
	Prop4	✓ (4.3s)	✗ (8.9s)	✗ (18.8s)	✗ (21.7s)	✗ (33.6s)	✗ (59.1s)	✗ (95.3s)

Table 1: Results for the 4 desirable properties for the 2 configurations of AgRob V16, including execution times. The startup configuration uses a forced scope of 8 Nodes, 7 Topics and 8 Fields. Regarding the map configuration, a scope of 11 Nodes, 12 Topics and 11 Fields was used.

compelling the generation of robust test-suites. During the employment of this technique, it's assumed that the nodes were properly tested and verified, and the subsequent intra-node specifications were defined.

For illustrative purposes, the Listing 5.1 shows an excerpt of the description file containing the inner-functional specification of the `SafetyController` node. The Listing 5.2 presents a second excerpt of the same file. Here, the five previously system-wide properties to be checked were translated to the HAROS specification language.

Notice that, the properties specification in HAROS change the focus from nodes to topics. Although each property is written within a specific node or configuration label, this it's only used to define the scope and the axiomatic possibility of the property itself. The focus on topics instead of nodes pops up as a requirement from the HAROS monitor dependency. This difference on focus is easily addressed through simple documentation inspection.

5.1.4 *Technique Evaluation*

Every property was checked for the two main configurations with increasing scopes for values and messages. Due to the system dimensions, the verification was bounded to the analysis of infinite recursive paths with, at maximum, ten different states. After executing the plug-in, every specified property has been checked with multiple scopes, for each configuration. The Table 1 shows the summarized results.

`Property 3` was actually shown not to hold for the `map` configuration. Commands to rotate in-place can happen, even when there is not an explicit indication by the joystick, or an obstacle detection by the lasers. This happens due to bad-configuration issues, that may create accumulated localization errors, which may cause wrong identifications of dangerous situations. Figure 18 shows how the counter-example is displayed in the HAROS to the user. The counter-example describes an execution trace, where the velocity command from

```
project: agrob

nodes:
  agrobv16_supervisor/agrobv16_supervisor_node:
     rosname: agrobv16SUPERVISOR
     hpl:
      properties:
        - 'globally: no /agrobv16/current_state {data[0] not in [0,3,6]}'
        - 'globally: /agrobv16/current_state {data[0] = 6}
            requires /joy_teleop/joy {button[0] = 1}'
        - 'globally: /agrobv16/current_state {data[0] = 3}
            requires /joy_teleop/joy {button[0] = 0, button[1] = 1}'
        - 'globally: /agrobv16/current_state {data[0] = 0}
            requires /joy_teleop/joy {button[0] = 0, button[1] = 0}'
        - 'globally: no /supervisor/cmd_vel {linear.x not in 0 to 10}'
        - 'globally: /supervisor/cmd_vel  {linear.x in 1 to 10 }
            requires /joy_teleop/joy {button[0] = 0, button[1] = 1} ||
                     /joy_teleop/joy {button[4] = 1} ||
                     /joy_teleop/joy {button[5] = 1}'
        - 'globally: /supervisor/cmd_vel {linear.x in 0 to 10,
            linear.x not in 3.8 to 5.8,
            linear.x not in 4.2 to 6.2}
                requires /joy_teleop/joy {button[0] = 0, button[1] = 1}'
        - 'globally: /supervisor/cmd_vel {linear.x in 3.8 to 4.2}
            requires /joy_teleop/joy {button[0] = 0, button[1] = 1} ||
                     /joy_teleop/joy {button[4] = 1, button[5] = 0}'
        - 'globally: /supervisor/cmd_vel {linear.x in 5.8 to 6.2}
            requires /joy_teleop/joy {button[0] = 0, button[1] = 1} ||
                     /joy_teleop/joy {button[4] = 0, button[5] = 1}'
        - 'globally: no /supervisor/cmd_vel {angular.x not in -100 to 100}'
        - 'globally: /supervisor/cmd_vel {angular.x not in -100 to 100}
            requires /joy_teleop/joy {button[0] = 0, button[1] = 1} ||
                     /joy_teleop/joy {button[4] = 1, button[5] = 0} ||
                     /joy_teleop/joy {button[4] = 0, button[5] = 1}'
        - 'globally: /supervisor/cmd_vel {angular.x in -8 to 12,
            angular.x not in -12 to 8}
                requires /joy_teleop/joy {button[0] = 0,
                    button[1] = 1} || /joy_teleop/joy {button[4] = 1,
                    button[5] = 0}'
        - 'globally: /supervisor/cmd_vel {angular.x not in -8 to 12,
            angular.x in -12 to 8}
                requires /joy_teleop/joy {button[0] = 0, button[1] = 1} ||
                         /joy_teleop/joy {button[4] = 0, button[5] = 1}'
        - 'globally: /agrobv16/max_velocity {linear.x = 0}
            requires /scan {ranges[0] in 0 to 4}'
```

Listing 5.1: Excerpt of the configuration file used to write the property specifications. It contains the functional inner-properties regarding the SafetyController node. These were extracted through the RoMoVi formal documentation.

```
configurations:
 startup:
    launch:
     -agrobv16_supervisor/launch/startup.launch
     hpl:
        properties:
            - 'globally: /agrobv16/current_state {data[0] = 3}
                requires /joy_teleop/joy {button[0] = 0, button[1] = 1 }'
            - 'globally: /agrobv16/current_state {data[0] = 6}
                requires /joy_teleop/joy {button[0] = 1}'
            - 'globally: /agrobv16/current_state {data[0] = 0}
                requires /joy_teleop/joy {button[1] = 0, button[0] = 0}'
            - 'globally: /husky_velocity_controller/cmd_vel {linear.x = 0,
                angular.x != 0}
                    requires /scan {ranges[0] in 0 to 4} ||
                    /joy_teleop/joy {button[0] = 1}'
            - 'globally: /husky_velocity_controller/cmd_vel
                {linear.x in 1 to 10}
                    requires /joy_teleop/joy {button[0] = 1}
                        || /joy_teleop/joy {button[1] = 1}'
```

Listing 5.2: Excerpt of the configuration file used to write the property specifications. Each specification is written on the field regard to the configuration under which the properties must be verified.

the `NavStack` depicts a dangerous situation that was not flagged by the lasers. This counter-example has required scopes of at least 4 values/messages to be detected. The result was displayed under 1min for scopes up until 10 values/messages.

`Property 4` has revealed a counter-example to both configurations, as expected. Figure 19 shows a screenshot of the HAROS interface. The image depicts the counter-example produced when verifying the property in the startup configuration. From this report is detected positive linear values on the Husky Interface, in such cases, where the emission of the expected values by the joystick are not required. At architecture level, this issue may be solved through the incorporation of a joystick filter, which disables the possibility to the Safety Controller receiving some robot commands prior to the controller activation. This counter-example was produced using at least 4 values/messages. Due to the smaller dimension of the startup configuration, this result was displayed under 1min for scopes up until 12 values/messages.

The times measured during the verification process shows empirical evidence of the technique practicability and utility when regarding safety-properties. All the properties were properly checked with up to 14 value/messages at around 2min, which represents an acceptable value if the technique is to be executed in continuous integration. Future work might focus on the times improvement through simple modifications, as the use of a distinct default model checker, or by resorting to the SMV verification techniques.

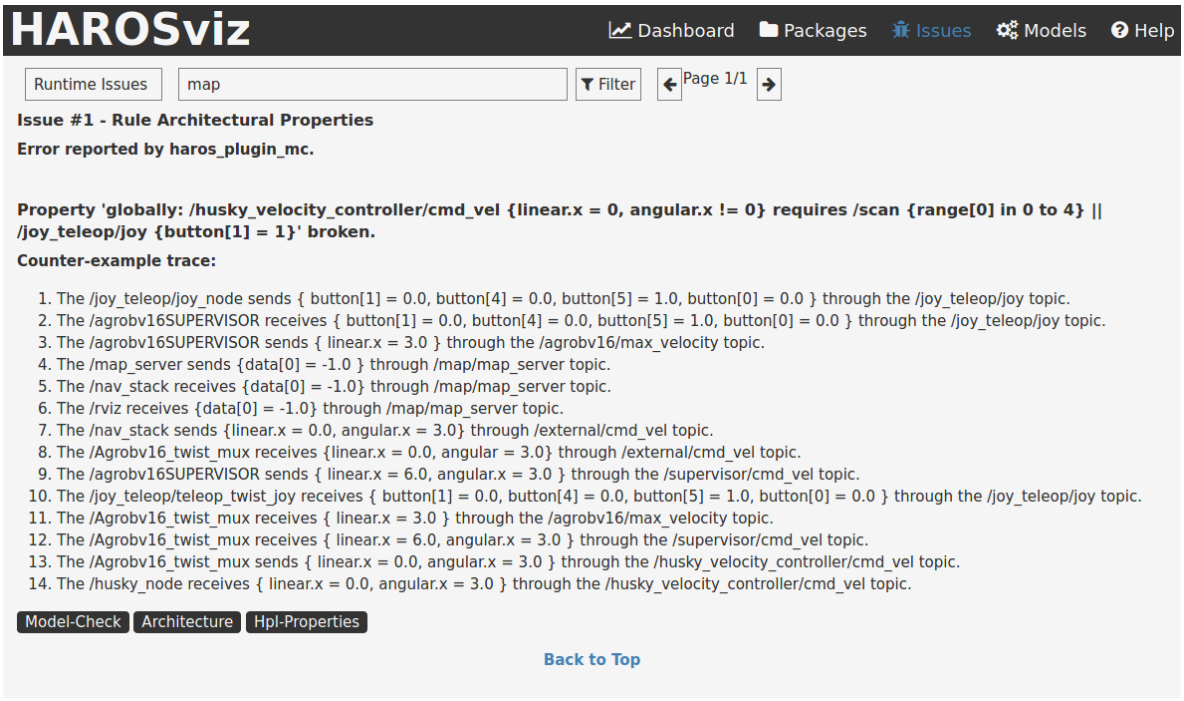


Figure 18: A counter-example to the third property at the map configuration. The counter-example is shown as a runtime issue based on concrete steps. Each step describes an action that can be monitored through the HAROS.

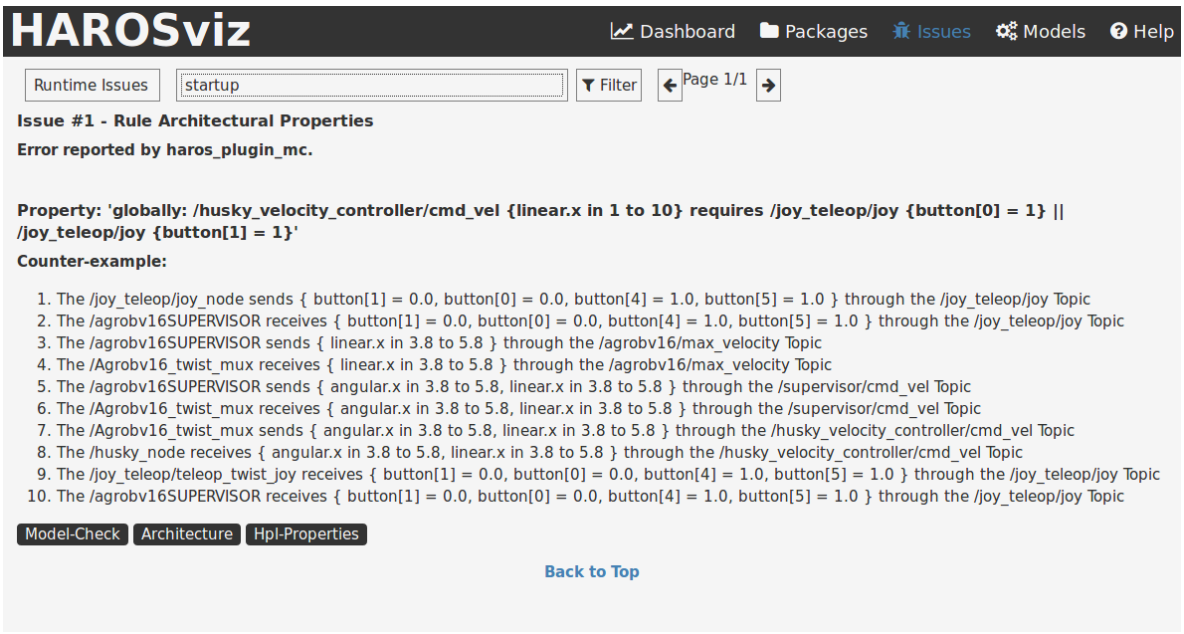


Figure 19: A counter-example to the fourth property at the startup configuration. The counter-example is shown as a runtime issue based on concrete steps. Each step describes an action that can be monitored through the HAROS.

As said before, the scope for `Message`, `Value` and **`Time`** must be carefully defined for each ROS repository. Small scopes will perform verification in small universes, which may hide possible safety issues. Being the approach intended to be practiced by the systems developer, the application-specific knowledge should be enough to infer sensible scopes for the most relevant traces. Another issue that was already mentioned, is the verification upon loose behavioural specifications, which may lead to false positive counter-examples. Although such cases were not identified during this study-case, they are expected to arise mostly when dealing with desirable liveness properties. This can be easily addressed by simply creating and executing a concrete test-case based on the produced counter-example.

6

CONCLUSIONS AND FUTURE WORK

This work has presented a technique to verify ROS system-wide safety properties through model-checking. It was based on the formalization of ROS launch configurations and loosely specified behaviour of individual nodes. The technique was wrapped in a HAROS plug-in, which extracts during continuous integration information regarding the systems structure from the configurations, as well as behaviours, from the PBT specifications. The plug-in automatically creates Electrum models upon which it performs bounded model-checking of the specified properties. When such properties do not hold, a Electrum counter-example is converted from its abstract form to a readable format within the ROS domain. These results are retrieved as HAROS issues through its common interface.

During this work, a novel Electrum idiom with automation potential on safety properties verification was proposed. In order to deal with scope issues, by resorting to interval analysis, an attempt to describe systems values through higher levels of discretization was successfully achieved. Furthermore, it has proved to be enough to describe and verify real systems safety properties. Despite the technique was only focusing ROS applications, the underlying theory and practice might be generalized for any modular architecture based on message-passing communication patterns.

After its application in a real industrial case, whose both, complexity and dimension are representative, the technique has proven to be sufficiently expressive to check certain classes of safety properties. The performance results have shown empirical evidence of its practicability under continuous integration environments. Besides that, the present work has originated a group publication, which at the moment, has already been reviewed and accepted by peers at an international venue [4].

Future work should focus in extending the support for richer property patterns, in particular, for scopes other than the global one. The complete integration of every HAROS language feature, excepting the real-time ones, seems to be a suitable aim. Furthermore, techniques to minimize the user intervention, as an automatic computation of optimal verification scopes may be introduced. Lastly, techniques to automatically discard false positives may be implemented, possibly by relying on run-time analysis to check the validity of the counter-example traces.

With regard to the property verification and its audience usability, this dissertation achieved its main goals, providing a reasonable technique for practicing automatic verification of system-wide safety properties. This may be extended during future works to incorporate the analysis of system-wide liveness properties.

Lastly, despite the positive evaluation results, additional inspection regarding the Electrum idiom complexity might provide useful information towards possible time optimizations.

BIBLIOGRAPHY

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[2] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pages 99–114. Springer, 2017.

[3] Patrick Blackburn, Johan FAK van Benthem, and Frank Wolter. *Handbook of modal logic*. Elsevier, 2006.

[4] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ROS applications. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page To Appear. IEEE, 2020.

[5] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[6] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[7] Filipe Neves Dos Santos, Heber Sobreira, Daniel Campos, Raul Morais, António Paulo Moreira, and Olga Contente. Towards a reliable robot for steep slope vineyards monitoring. *Journal of Intelligent & Robotic Systems*, 83(3-4):429–444, 2016.

[8] Matthew B Dwyer, George S Avrunin, and James C Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.

[9] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[10] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Formally introducing Alloy idioms. In *Proceedings of the Brazilian Symposium on Formal Methods*, pages 22–37, 2007.

[11] Raju Halder, José Proença, Nuno Macedo, and André Santos. Formal verification of ROS-based robotic applications using timed-automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50. IEEE, 2017.

[12] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[13] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. Phys: probabilistic physical unit assignment and inconsistency detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 563–573, 2018.

[14] Fred Krger and Stephan Merz. Temporal logic and state systems. *Texts in Theoretical Computer Science. An EATCS Series.*, 2008.

[15] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 45–48, 2002.

[16] Nuno Macedo and Alcino Cunha. Alloy meets TLA+: An exploratory study. *arXiv preprint arXiv:1603.03599*, 2016.

[17] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 373–383. ACM, 2016.

[18] Niloofar Mansoor, Jonathan A Saddler, Bruno Silva, Hamid Bagheri, Myra B Cohen, and Shane Farritor. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 785–790. ACM, 2018.

[19] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[20] Kenneth L McMillan. The SMV system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[21] J. Mendes, F. N. d. Santos, N. Ferraz, P. Couto, and R. Morais. Vine Trunk Detector for a Reliable Robot Localization system. In *2016 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–6, May 2016. doi: 10.1109/ICARSC.2016.68.

[22] Joseph P Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 31–40. ACM, 2011.

[23] Jason M O'Kane. A gentle introduction to ROS, 2014.

[24] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Lightweight detection of physical unit inconsistencies without program annotations. In *Proceedings of the 26th*

ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 341–351, 2017.

[25] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. Phriky-units: a lightweight, annotation-free physical unit inconsistency detection tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 352–355. ACM, 2017.

[26] John-Paul Ore, Sebastian Elbaum, and Carrick Detweiler. Dimensional inconsistencies in code and ROS messages: A study of 5.9 m lines of code. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 712–718. IEEE, 2017.

[27] Rahul Purandare, Javier Darsie, Sebastian Elbaum, and Matthew B Dwyer. Extracting conditional component dependence for distributed robotic systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1533–1540. IEEE, 2012.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[29] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the robot operating system*. O'Reilly Media, Inc., 2015.

[30] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. A framework for quality assessment of ROS repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496. IEEE, 2016.

[31] André Santos, Alcino Cunha, and Nuno Macedo. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 56–62, 2018.

[32] André Santos, Alcino Cunha, and Nuno Macedo. Static-time extraction and analysis of the ROS computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69. IEEE, 2019.

[33] André Santos, Alcino Cunha, and Nuno Macedo. Seabass: System and behaviour abstraction with short specifications. Submitted.

[34] Nishant Sharma, Sebastian Elbaum, and Carrick Detweiler. Rate impact analysis in robotic systems. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2089–2096. IEEE, 2017.

[35] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[36] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[37] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Transactions on Human-Machine Systems*, 46(2):186–196, 2015.