Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

January 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Manuel Alcino Pereira da Cunha
André Filipe Faria dos Santos

January 2022

ACKNOWLEDGEMENTS

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

Industrial manufacturing is becoming highly reliant on automation developments, as they bring more efficient and accurate processes with less associated cost. Consequently, robots are increasingly being deployed in a wide range of scenarios, especially where safety is demanded. In such cases, it is critical to employ appropriate procedures to verify both the system's quality and safety.

Following the current growth of cyber-physical system, as well as their usage in various technology domains, the development of software applications is becoming more demanding due to the complexity behind the integration of needed services, beyond those provided by the operating system. Therefore, software middleware is increasingly used, since it offers services that support application development and delivery.

One of the most popular open-source software platforms for building robotic systems is the Robot Operating System (ROS) [56] middleware, where highly configurable robots are usually built by composing third-party modules. A major factor behind its popularity and widespread adoption is its flexibility and interoperability. One drawback of this flexibility, however, lies in the increased security risks that ROS applications face. The emergence of performance and scalability challenges connected to the ROS middleware standard, in addition to security concerns, prompted the creation of ROS2.

Robot Operating System 2 (ROS2), which continues to provide a simple, uniform message passing interface, to allow components to communicate with each other, is implemented using the Data Distribution Service (DDS) [52] communication protocol, where security guarantees are ensured by the DDS-Security specification. Using DDS-Security, it is possible to configure ROS2 to run with security guarantees using the SROS2 toolset [28].

This dissertation will propose a technique, based on the software verification perspective, to automatically verify system-wide properties related to the security configuration of ROS2-based applications. To that purpose it will model the ROS architecture, as well as the network communication behaviour, in Alloy [32], a formal specification language and analysis tool supported by a model-finder, with which system-wide properties will subsequently model-checked.

KEYWORDS Robotics, ROS, ROS2, DDS, SROS2, Security, Software Verification, Alloy

RESUMO

A crescente implementação da automização de processos tem motivado a reestruturação nos mais diversos setores industriais, com o objetivo de aumentar a eficiência e precisão nos processos, e consequentemente, reduzir os custos associados. Além disso, esta ideia levou à integração da robótica nos mais amplos domínios tecnológicos, especialmente onde a segurança é exigida. Nestes casos, é fundamental adotar técnicas apropriadas de forma a verificar tanto a qualidade do sistema, como a segurança do mesmo.

Como resultado do atual crescimento dos sistemas ciber-físicos, nomeadamente sistemas robóticos, bem como a sua utilização em vários domínios tecnológicos, o desenvolvimento de aplicações tem vindo a ficar mais exigente, em particular devido à complexidade da integração dos serviços necessários, tipicamente não fornecidos pelo sistema operativo. De forma a acompanhar o aumento da complexidade destes sistemas, *middlewares* têm sido adoptados, pois integram serviços que oferecem suporte ao desenvolvimento de aplicações robóticas.

Uma das plataformas considerada como *standard* no que toca ao desenvolvimento sistemas robóticos é o *middleware* Robot Operating System (ROS) [56], onde robôs altamente configuráveis são construídos atráves da composição modular de *software* externo, oferencedo características como flexibilidade e interoperabilidade aos sistemas integrados. No entanto, esta constante priorização na flexibilidade resulta num aumento de vulnerabilidades de segurança, pondo em causa a integridade das aplicações. Além da falta de segurança, existem também problemas de desempenho e escalabilidade relacionados com a especificação do *middleware*. Assim, era necessário uma mudança na estruturação do ROS, resultando na criação do Robot Operating System 2 (ROS2).

O Robot Operating System 2 (ROS2) implementa um protocolo de comunicação, de nome Data Distribution Service (DDS) [52], que para além de garantir serviços de comunicação, fornece diversas especificações. A especificação DDS-Security, que através de uma metodologia de *plugins*, oferece diferentes métodos de adoção de segurança. Através do uso desta especificação, juntamente com o uso do *toolset* SROS2 [28], é possível configurar o ROS2 de forma a propocionar um ambiente seguro às aplicações integradas.

Esta tese irá propor uma técnica para a verificação automática de propriedades globais de segurança em aplicações ROS. Esta técnica irá basear-se na formalização estrutural de arquiteturas ROS em Alloy [32], assim como no comportamento associado à comunicação dentro do sistema, por forma a permitir a verificação das desejadas propriedades de segurança.

PALAVRAS-CHAVE Robótica, ROS, ROS2, DDS, SROS2, Segurança, Verificação de Software, Alloy

CONTENTS

LIST OF FIGURES

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

Automation is increasingly being incorporated into the industrial world, through the use of flexible tools to assist in the most various scenarios, as it brings efficiency and accuracy to the industry's processes. Robotics is already the key driver of competitiveness and flexibility in large scale manufacturing industries. Due to the continuous growth of technology in these different domains, robots can be used in a wide range of applications [50], since their usage brings increased productivity, safety, and more manufacturing production work back to developed countries [27, 19].

Despite the advances in technology, dealing with hardware-level applications becomes highly impractical as the system's complexity increases. Therefore, developing and writing software code for robot applications is becoming more demanding, namely because multiple aspects must be properly considered [55].

Since robots are essentially integrated as distributed systems with separate components, the need to connect different hardware and software modules raises interoperability and communication issues. To solve these issues, modular architectures, based on message-passing communication patterns, are continually emerging as the architecture's middleware layer. Their primary focus is to offer services to the application layer, consequently easing the development cost, while providing interoperability and communication facilities [50, 48]. The requirement for a middleware layer that meets different robot's specification is a novel approach to enable the creation of robotic applications, while supporting features such as robustness and modularity.

The Robot Operating System was created by a collaborative open-source community to contribute to the advancement of robotics, in particular with the aim of helping build robotic applications easily [26]. It provides locomotion, manipulation, navigation, and recognition tasks via third-party software libraries and tools. Concerning the wide range of robotics hardware and software, ROS was designed to be flexible, enabling the easy integration of external components. However, performance and scalability issues arise due to its middleware specification [55]. Additionally, real-time constraints such as fault-tolerance, deadlines, or process synchronization were not supported by ROS, making it unsuitable for developing safety-critical and real-time systems [38].

Besides having no middleware support for distributed real-time systems, security was not prioritized in ROS, which started to be problematic for deployed systems. An increasing number of real-time applications, for instance robotic systems, requires security for protecting real-time sensitive data against unauthorized access [42].

This led to the creation of the Robot Operating System 2, developed using the Data Distribution Service (DDS) [52] specification protocol as its communication middleware. Issues concerning system integration and scalability are mitigated by DDS various implementations, due to the several transport configurations provided, making it

suitable for real-time distributed systems. DDS also provides a security specification, called DDS-Security. ROS2 makes use of this specification, to provide security guarantees to the deployed robotic systems [5].

Due to the widespread usage of robotic systems, software verification, through the use of formal methods, are necessary to prevent potentially catastrophic consequences, mainly related to security matters, as safety guards are gradually implemented into the software domain [73]. Within this context, the Alloy [36, 44] framework enables the behavioural representation of systems with rich configurations, due to the combination of both relational and linear temporal logic (LTL) provided by its specification language, consequently supporting model-checking techniques. Model-checking techniques enable far better levels of coverage and, as a result, more reliability, than traditional testing. In model-checking the system is abstracted in a formal model, that is automatically checked by performing property verification on finite-state machines [6].

The proposal of this dissertation is to develop a novel technique to automatically verify system-wide safety properties of ROS2 applications using Alloy framework. The idea will be to represent a ROS2 system into an abstract Alloy model, to be model checked, and obtain a prototype tool that can be used by developers to easily detect security configuration issues on their respective robotic application.

1.1 OBJECTIVES AND CONTRIBUTIONS

The first goal of this thesis is to survey the concepts around the Robot Operating System, contextualizing the evolution behind its framework towards achieving security, which the former version of ROS lacked due to the focus on flexibility.

Since ROS2 has been developed over the DDS framework as its communication middleware, DDS must be properly understood before considering the security aspects.

The DDS Security standard functionality will be presented, as well as how security is integrated into ROS2. Since security issues, concerning public networks, are recent to the robotics domain, ROS2 security network design should be analyzed structurally via access control files configuration.

The second goal is to extend a previously proposed [11] formalization of ROS applications in Alloy [32, 44], to also take into consideration the security configuration defined with SROS2. Using this extension, the viability of verifying simple information-flow security properties will be explored. For instance, to ensure that no commands to the vehicle motor can be sent via the infotainment system.

The final goal is to automate the extraction of such formal Alloy models from the configuration files of a ROS2 application, in order to obtain a prototype tool that can be used by roboticists to easily detect security configuration issues.

1.2 DOCUMENT STRUCTURE

The reminder of this document is divided into *3* different chapters, where the first *2* regards the state of the art (2, 3) followed by the future dissertation plan (4).

Chapter (2) introduces the Alloy framework, presenting its specification language with the help of a concrete example.

Chapter 3 introduces all the concepts related to Robot Operating System, and its evolution as robotic development framework towards achieving system security. Subsequently, the Data Distribution Service and its security specification architecture are mentioned and studied. In section 3.3, some current works that fit the same domain of this dissertation are overviewed.

Finally, in chapter 4, expected considerations about the future work are outlined, where a timetable schedule for the remaining work is also presented.

2

THE ALLOY SPECIFICATION AND ANALYSIS FRAMEWORK

As aforementioned, this dissertation aims to tackle the security vulnerabilities resulting from the miss-configuration of ROS files. In this chapter, it is intended to explore the Alloy framework that will be used to overcome the above-mentioned challenge.

The increasing usage of robotics in safety-critical systems requires ensuring the proper correctness of both software and hardware, as failures often lead to fatal consequences, namely regarding the security domain. Thus, the use of formal methods and verification techniques, especially in systems highly reliant on flexibility and reliability, is recommended to avoid security-critical faults [14]. Software frameworks designed for this purpose must provide methods to perform structural design over systems with rich structures, abstracting their behaviour as a conventional formal model. Additionally, these frameworks must support features to enable automate analysis, in which property verification over these designed models is used as technique.

The *Alloy Framework* [36], fits within this context, as it provides a declarative relation-based language, used for software modelling, complemented with extensive tool support for analysis over these models [32]. The language combination of both *Relational* and *Linear Temporal Logic* (LTL) enables the ability to model both systems with rich structures and complex behaviour. To address the correctness of the specified model, Alloy performs model-checking techniques over these logic languages, where a model M is exhaustively checked over property verification [44].

The framework analyzer, the Alloy *Analyzer*, takes the specified model's restrictions into account and performs *Bounded* and *Unbounded* Model Checking to find instances that satisfy a given property. It can be also be used for checking model properties, where the analyzer will try to return a counterexample instance. Instances are displayed by the framework *Visualizer*, including a step-by-step representation of traces. Instance appearance can be customized using themes [32].

This chapter will go through these principles in further depth, to give the reader a proper understanding on how Alloy is structured, supported by a previously configured example where the paradigm behind *Eurelian* Circuits is structurally modelled. Since system analysis rely on Model-Checking techniques, the following section will briefly introduce a clear contextualization on this matter.

2.1 MODEL CHECKING

Performing software testing has been regarded as the established assessment procedure to check functional and non-functional specifications. The conventional approach for software verification is based on testing the system with different inputs, to achieve quality assurance over several intended specifications [6, 8]. As this technique demands exhaustively evaluation over pre-selected test data, it is commonly applied with automated tools, since manual testing is time-consuming and prone to errors [15, 30].

Model Checking presents itself as a remarkable technique with the purpose of verifying temporal properties over a system finite-state, with the latter being duly represented as a conclusive model. Additionally, it enables *model-based testing* by automatically interpreting counterexamples as test cases, resulting in significantly greater degrees of coverage than conventional testing [30, 6]. This technique is becoming increasingly used due to its importance as an early phase verification technique when developing systems [44].

It provides highly automatic verification procedure, unlike other techniques, such as theorem provers, which are based on deductive reasoning. The fact that false specifications yield counterexamples, confers great functionality to this technique, and is essential for debugging matters. However, the inevitable state explosion causes a complexity increase on verification. This is referred to as the *state explosion problem*, occurring when model-checking is unable to handle the size of the state space [14, 13]. Yet, this is can be mitigated using bounded techniques, that limit the exploration of the finite-state model to a given depth.

Model Checking techniques enable property verification of systems through the implicit use of temporal logic to express dynamic behaviour through the course of the system evolution. Thus, the system is abstractly represented as a transition system, and property checking is performed over the latter, considering a formula defined in a temporal logic [34, 65].

TRANSITION SYSTEM A *Transition System* is defined as a graph-based representation of a system dynamics, that confers additional representation over the mathematical graph structure. For model checking purposes, transition system must be equipped with a labelling function that maps each state to the set of *atomic propositions* that are true at that state. These propositions describe the values of the system variables in each state [51, 65].

A transition system with such labelling function essentially corresponds to a *Kripke Structure* [51]. Conceptually, a *Kripke Structure* corresponds to a model M with the following tuple structure $M = (S, I, R, p, L)$, where: S is a finite set of states; I represents a set of initial states, so naturally I is a subset of S ($I \subseteq S$); R defines the *transition relation* as it accounts the transitions between states; L is an *interpretation* that defines the labelling function; Accordingly, it assigns each state with a set of valid atomic propositions $L(s)$ that are valid in it, drawn from the domain p ($L(s) \subseteq p$).

Model Checking is a *model-based* technique where verification focuses on the concept of property satisfaction. Therefore, the corresponding model-checker must be capable of checking if the model M satisfies a desirable property, expressed as temporal logic formula ψ, in a model state s: $M, s \models \psi$ [34, 51].

The semantics of the temporal logic relies on how the latter addresses time as an evolving approach towards state verification. Notably, temporal logics are either qualified as *linear-time* or as *branching-time* [34]. In the former approach, time is perceived as linear path and the corresponding transition system is abstracted by a set of infinite traces. The latter denotes time as a branching model, in which the transition system is abstracted by a set of infinite computation trees, consequently enabling non-deterministic considerations about the system evolution. The choice on the logic semantics depends on the system properties to be analyzed [51], and entails different model-checking algorithms [34].

2.2 MODELLING

The Alloy framework presents itself as a formal modelling language for both structural and temporal behaviour design, where properties are defined over their specification language. Formerly, Alloy was inherently static [44], meaning that it only excel the structural design, with a specification language based on first-order relational logic. The analysis process relied on a bounded model finding technique with no support for temporal behaviour. Notwithstanding, the latest release of Alloy[1] confers the ability to properly deal with expressive temporal properties, as well as trace evaluation over time [32].

As intentionally design to formally abstract both system's configuration and behaviour, Alloy successfully incorporates a set of features, within a well-documented and wide-ranged syntax that consequently allows large specification development [44]. The following subsections present the Alloy concepts required for understanding how system modelling is performed.

2.2.1 *Structural Modelling*

Alloy aims to address the complexity behind richly structured systems, that require critical control over their intended behaviour. Here, system's structures can be specified over time-evolving states, where its behaviour clearly identifies the states' in-between transitions. The conception of system transitioning offers a great formal approach when it comes to reason about the system's design.

The Alloy structural definition relies on a relation way of connecting system's elements, where structures are modelled with relations. Unary relations, commonly known as sets, are denoted as *signatures* in Alloy. Signatures are inhabited by a set of *atoms*, drawn from a finite universe of discourse. Atoms are perceived as the lowest-grain elements, with no particular semantics attached. A signature, identified by the keyword sig, might include multiple *field* declarations enclosed between braces, to declare relations between the signature's atoms and other signatures' atoms. Fields are inhabited by tuples of atoms from the universe, that must have the same arity.

Signatures can either be perceived as a top-level signature, or as other signature's subset. Signature hierarchy is specified through disjoint extensions ($extends$), or by set inclusion (in). The $abstract$ keyword declares a signature that contains no atoms beyond those within its extensions.

1 https://alloytools.org/alloy6.html

To address default configuration over the universe's multiplicity, both signatures and fields can be specified under a multiplicity constraint. The former constrains the number of signature atoms, and is commonly used to express singleton sets, with the keyword $one\ sig$. Fields, however, makes great use of multiplicities by restricting behaviour over relations between atoms. In addition to these implicit constraints, system assumptions can be defined with axioms, expressed as *facts*, where multiple constraints can be incorporated.

Throughout the sections that follow, it will be presented an illustrative example from graph theory, this being the study of *Eulerian Circuits*. This example will be used to duly contextualize both modelling and verification process in Alloy. Eulerian Circuits must meet several behaviour constraints over the classic graph definition, that must be addressed over model constraints. However, the structural model must be provided beforehand.

Considering the Alloy's abstract ability to reduce complexity over model designing, at a high degree of abstraction, graphs can be represented as a set of nodes, connected together with a relation, with no need to address edges as a separated structure declaration.

```
sig Node {
    adj : set Node,
    var visited : set Node
}
```

Graph representation over a sigle Node declaration.

As depicted above, the sig keyword is followed by the corresponding *Node* signature declaration. The *Node* signature is defined by a static set of node atoms, that combined denotes the finite universe of discourse.

Then, fields are enclosed between braces. The *adj* relation models the graph edges, where each node can be connected to a set of nodes. As it is identified as an immutable field, the corresponding relation between atoms is static. Moreover, addressing additional graph functionality, it is desirable to concern about the visited nodes.

Relation multiplicity constraints are explicitly defined in the field declaration through the use of multiplicity operators, those being one, $lone$, $some$ and set, with the obvious semantics. The field multiplicity operators could be used to limit the number of signature atoms. Despite this, signature multiplicity is frequently used to represent singleton $one\ sig$ sets.

```
one sig Init extends Node {}
var one sig Euler in Node {}
```

Node signature hierarchy.

Both signatures are preceded by the one keyword, imposing a multiplicity constraint over each signature declaration. Setting the multiplicity to one means that each assessed model instance must have precisely one *Init* atom and one *Euler* atom. It should be noted that, since the *Node* declaration is not preceded by the $abstract$ keyword, its atoms do not solely belong to the *Init* signature.

Modelling constraints can be specified by making use of the $fact$ declaration. The formula specified inside each $fact$ declaration denotes a model axiom, that holds a truth model assumption, to serve as a premise for further reasoning.

The Eurelian path denotes a trail within a finite graph, with each graph edge being visited precisely once. Thus, this implies that the graph must be connected, where each node must be reachable from each other, and undirected. Moreover we will assume that the graph has no self-loops.

```
fact eulerian_considerations {
    adj = ~adj
    no iden & adj
    all x : Node | x->Node in ^adj
}
```

Graph restrictions through $fact$ declaration.

These constraints, specified above, make use of some Alloy operators, which can either be identified as a set-theory operator or as a relational operator (Figure 1). Despite the extensive number of operators supplied by Alloy's language, every expression, through the usage of *FOL* quantifiers and *LTL* operators, is later translated to boolean-based expressions.

Alloy	Math
Φ in Ψ	$\Phi \subseteq \Psi$
Φ = Ψ	$\Phi = \Psi$
lone Φ	$\lvert\Phi\rvert \leq 1$
some Φ	$\lvert\Phi\rvert \geq 1$
no Φ	$\lvert\Phi\rvert = 0$
one Φ	$\lvert\Phi\rvert = 1$

Alloy	Math
iden	id
Φ + Ψ	$\Phi \cup \Psi$
Φ & Ψ	$\Phi \cap \Psi$
Φ - Ψ	$\Phi \setminus \Psi$
Φ -> Ψ	$\Phi \times \Psi$
Φ . Ψ	$\Phi \,.\, \Psi$
A <: Φ	$A \lhd \Psi$
Φ :> A	$\Phi \rhd A$
~ Φ	Φ°
^ Φ	Φ^{+}
* Φ	$\Phi^{\star}$
{ x : A | ϕ }	$\{x \mid x \in A \wedge \phi\}$

Figure 1: Relational Logic Syntax.

Regarding set-theory, this fact uses set intersection (denoted by $\&$) and set union (denoted by $+$). The first-order logic syntax can also be used, since it provides a more user-friendly way of specifying expressions. The last constraint makes use of the universal quantifier all with a respective expression (*x : Node* which mathematically represents the predicate $Node(x)$). The formula is then specified after the $|$ notation.

The remaining are relational operators. The $\sim adj$ denotes the converse relation of *adj*, , and the $\rightarrow$ represents the Cartesian product operator. The positive transitive closure operator (^) yields the smallest transitive $\hat{}adj$ relation containing all the pairs of tuples, reachable in one or more steps, through the implicit use of the set composition operator ($\cdot$). The reflexive transitive closure $*$ is defined as $*adj = \hat{}adj + iden$.

$$\hat{}adj = adj + adj \cdot adj + adj \cdot adj \cdot adj + ...$$

Figure 2: The positive transitive closure operator.

In addition to having explicit constraints defined in $fact$ axioms, structural constraints can also be implicitly defined in a signature declaration. These are often referred as signature facts, universally quantified over the signature's set [3]. For example, consider the following graph property, where it is desirable to express the following axiom: *The graph contains no self-loops.*

```
sig Node {
    adj : set Node,
    var visited : set Node
} {
    this not in adj
}
```

No self-loop constraint defined in the signature definition.

The latter expresses the *no self-loop* graph property. Even though *adj* is introduced as a relation between *Node*s, it is perceived as a set of *Node*s due to specification with the signature declaration. The keyword $this$ addresses each *Node* atom, as self representation of the considered atom. The remaining specifies the intended behaviour, as it specifies the non-inclusion ($not\ in$) over the set of its adjacent nodes.

Now that it has been specified both modelling structure and its constraints, it is advisable to run our model, through a process that will be later explained, in order to check that no unwanted model structural scenarios are presented. Since every model constraint was duly specified, in the Figure 3 is depicted an instance of an acceptable configuration for a *Eulerian* graph with *5 Node* atoms.

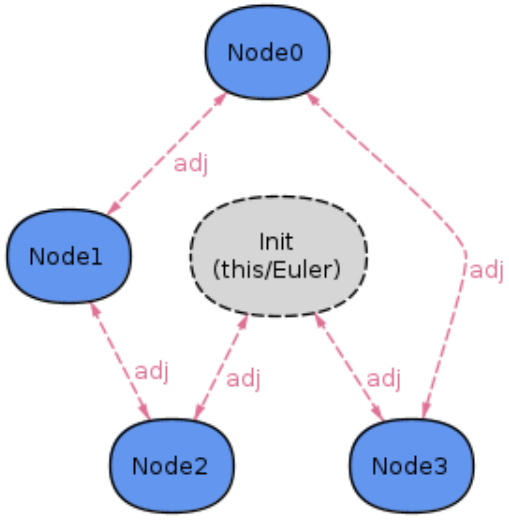

Figure 3: Acceptable *Eulerian* graph model design.

2.2.2 *Behavioural Modelling*

Model's behaviour representation represents the ability to express what is intended to happen during state transitions over the valid traces of a system. A *trace* is represented as an infinite chain of states that completely describes a system's potential behaviour. Valid traces are constrained by the explicit specification of axioms [31].

Despite the usefulness in restricting the system's traces, it is not ideal to express every property as a model assumption. Alloy high level of expressiveness enables the modelling of behaviour with multiple event idioms [44].

Moreover, the latest Alloy version allows the value of relations to change throughout the trace evolution, consequently allowing the declaration of both signatures and fields as time mutable declarations, through the usage of the keyword var. The notion of having mutable expressions implies extra care upon specifying events, since their evaluation might evolve to a non-wanted scenario.

Recall that the *visited* signature field that must be represented as a mutable relation, meaning that its evaluation may change during the course of the trace's evolution, as opposed to the static ones. The *Euler* node is also accounted as variable, since it can be included in an arbitrary atom.

The most common is to use a temporal formula that restricts the valid transitions to a set of events, where events are conveniently specified in separate *predicates*. The latter, declared with keyword $pred$, express boolean formulas that only hold when invoked. The notion of reusable expressions (fun) and model assertions ($assert$), are also conceptually included in the Alloy's event idiom. This idiom flexibility allows the developer to define action hierarchy, alongside with sharing atoms passed as parameters [44, 9].

Generally, events are specified with their respective event *guards* and event *effects*. A guard specifies a formula that must be true prior to the occurrence of the related event. Oppositely, an effect specifies how the system evolves to the next state, providing a valid outcome of its event. The effect must refer to the mutable variables of the model through the usage of the $'$ operator, to guarantee the non-occurrence of unexpected behaviour. Alternatively, the temporal operator $after$ can be used to ensure the truth of a formula in the next state.

The key event in our example is the *traverse* predicate, where the singleton signature *Euler* moves to an adjacent node. This event can be specified as follows.

```
pred traverse {
    some adjn : Euler.adj {
        adjn not in Euler.visited
        visited' = visited + Euler->adjn + adjn->Euler
        Euler' = adjn
    }
}
```

Eulerian visiting event.

The first guard requires the existence of some node *adjn*, that is adjacent to the *Euler* node (The set of nodes adjacent to Euler can be specified using the composition operator as $Euler \cdot adj$). The other guard ensures that each edge must be visited exactly once ($adjn\ not\ in\ Euler \cdot visited$). It is followed by *2* effects, the first updating the *visited* relation to include the latest visited edge, and the second setting the new node for *Euler*.

Unexpected behavior can occur if no constraints are set on how the system grows through such mutable expressions. Constraints imposed on mutable expressions that should remain unchanged in the next state are referred as *frame conditions*, a formula with primes (′) stating the "no change" effect [9].

It is advisable to consider the "nothing changes" possibility in the specification of the trace evolution. The following *stutter* predicate captures such "no effect" event and consists only of frame conditions.

```
pred stutter {
    visited' = visited
    Euler' = Euler
}
```

Eulerian stutter event.

Every conceivable trace of the system must be a permutation of these event performed on a well-defined initial state. When used consistently, this approach constitutes a model design pattern called *Implicit Operation Idiom* [31].

The following *_init* predicate states valid conditions in the first state: Every edge must start unvisited ($no\ visited$) and the *Euler* node should start in the *Init* node.

```
pred _init {
    Euler = Init
    no visited
}
```

Eulerian Initial State.

As previously claimed, to reason about state transition, the notion of an execution trace should be duly introduced. The system's permissible behavior can be thoroughly specified by restricting the set of valid operations. This can be done with the following axiom.

```
fact traces {
    _init
    always (traverse or stutter)
}
```

Trace constraint through the use of an axiom.

The *always* temporal operator (Figure 4) expresses a universal quantifier over time, imposing a constraint throughout the trace. Here, the operator is followed by the desired formula to impose a constraint that must be valid in every state, in this case that one of the *2* events must always occur. This fact also invokes to *_init* predicate to ensure the proper initialization of the system.

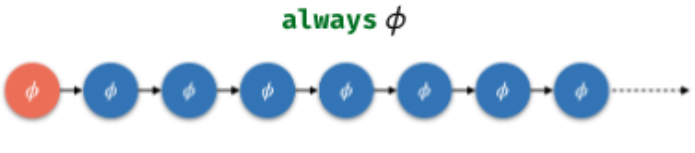

Figure 4: $always$ behavioural representation.

The behavioural modelling process clearly denote the idea behind the *Kripke* model tuple structure. The set of initial states (I) are imposed by the predicate *_init*, where the constraints defined in the latter represent, in some way, some p_i atomic propositions. Then, the transition relation R accounts both predicates *traverse* or *stutter*, which denote the transitions between states along the trace. Each state s ($s \subseteq S$) is defined by a set of atomic propositions p_s ($L(s) = p_s$), which each proposition is constrained by both initial propositions p_i and the transition behaviour.

2.3 ANALYSIS

Structural modelling only confers a conventional way of formally expressing the intended behaviour over a software component. To perform verification over the specified behaviour, it is advisable to implement analysis techniques, while supporting an illustrative way of exploring the behaviour through simulations.

Following the temporal logic semantics, Alloy specification language embeds the linear temporal logic into the first-order logic, thus, it makes use of both temporal and relational quantifiers to properly express behavioural verification over time [44]. *First-Order Temporal Logics* present additional techniques for reasoning about behaviour, while accounting the basis of the first-order logic, that usually confers the capability to express the well-formedness of the system structure [44, 40].

This section aims to explain how Alloy conducts system analysis, with further explanations over the corresponding analysis commands, along with an overview on the Alloy Analyzer interactive exploration process.

2.3.1 *Analysis Commands*

The specification language holds *two* analysis commands, run and $check$ respectively. A temporal logic formula is enclosed between braces inside a command, to be checked over. Due to Alloy's implicit abstraction, both commands addresses the relational model specification as truth holder, with the latter being consisted by declarations over facts and signatures.

Briefly speaking, a run command instructs the model-checker to present an example that satisfies the enclosed formula ψ_f as well as the model specified in the remaining facts (denoted by M). This means that the formula $M \wedge \psi_f$ is expected to hold in the returned instance. The consistency of the facts and signatures is consequently also verified since the model declaration M is also considered.

Alloy makes use of the $check$ command to perform automatic verification over an $assertion$ declaration. The assertion ψ_f verification is done by checking that formula $M \models \psi_f$ is valid within the defined scopes (Figure 5).

As result of the first-order logic's undecidability problem, the consistency proofness over satisfiability formulas is not possible [65]. To ensure decidability, the Alloy Analyzer performs analysis with *scopes* assigned to each signature declaration. By default, if no scope is provided, the model-checker will consider at most *three* atoms for each top-level signature.

Since scopes imposes a limit on the state space explored by the Alloy Analyzer, if no instance is returned, the formula can not be for sure considered inconsistent (in run commands) or valid (in check commands), since a bigger scope could allow the command to return an instance. However, due to the small scope hypothesis [36] it is likely that most commands can be satisfied with small scopes.

check { ϕ } **for** ... **but** k **steps**

Figure 5: Temporal formula ψ verification using *check*, accounting the Bounded model checking technique.

Here, the Alloy Analyzer tries to search for a *lasso trace* instance that refutes the formula $M \models \psi_f$ over the *check* command, by trying to find an instance that satisfies $((M \wedge \neg\psi_f)$. If such an instance is found, it denotes a counter-example to ψ_f. This technique is called *Proof by Refutation* (Figure 6): M entails ψ_f, denoted by $M \models \psi_f$, *if and only if* $(M \wedge \neg\psi_f)$ is unsatisfiable, reducing validity to unsatisfiability.

Moreover, Alloy yields a command to specify the finite number of different steps, related to the expected instance's evaluation trace. This is due to the bounded nature of the Alloy transitions between states, where the Bounded Model Checking is considered as model-checking technique. The number of considered steps is set to *10* by default, although this may be adjusted using the keyword *steps*, alongside with the signature scopes specification.

run { **not** ϕ } **for** ... **but exactly** 1 **steps**
run { **not** ϕ } **for** ... **but exactly** 2 **steps**
...
run { **not** ϕ } **for** ... **but exactly** k **steps**

Figure 6: Reducing Validity to Unsatisfiability.

The following run example makes use of both scopes and steps definitions. Moreover, the specified formula states the expected behaviour of an *Eulerian* circuit.

```
run example {
    eventually (adj in visited and Euler in Init)
} for exactly 5 Node, exactly 5 steps
```

Bounded Model Checking: Eventually the graph will represent an Eulerian circuit.

The *eventually* operator (Figure 7) expresses an existence quantifier, imposing the formula verification somewhere along the trace evolution [9]. Here, is used to specify the desired final state, where every edge is visited and the *Euler* node finishes where it started.

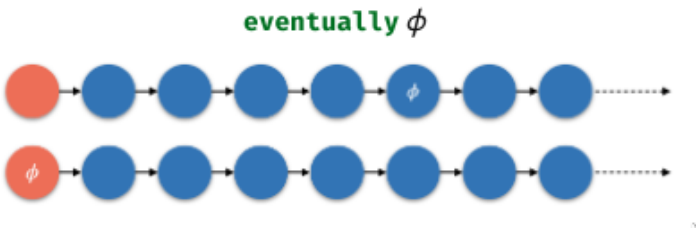

Figure 7: *eventually* behavioural representation.

This particular *run* yields no instance, since imposing *5* steps is not sufficient to yield an instance where the desired final state is reached.

Checking Behavioural Properties

The verification process over *Model Checking*, needs to account the formal specification of properties that are relevant to reason about the system's temporal behaviour [4]. Here, Alloy includes temporal connectives from FOLTL semantics, that acknowledges the system's states along the trace evolution [44, 32, 11].

Ensuring the correctness of a system's behaviour relies on verifying properties regarding the latter's *Safety* and *Liveness*. The former verification requires an invariance argument, where the latter verification requires a well-foundedness argument to prove its system satisfiability [2, 39].

A *safety* property asserts that "nothing bad should happen" during the system execution. Consequently, if a trace that violates a safety property is found, it can be assumed that the latter has a "bad" property prefix.

As examples of safety properties, consider the following assertions. The first denotes the implicit consideration about the evolution of the relation *visited*, which can only evolve through time.

```
assert safety_visited {
    always visited in visited'
}
```

Safety Property: The relation *visited* can only evolve through time.

The second denotes that if a certain node is visited, the *Euler* node must have been 'inside' it in the past. This assertion makes use of the *once* past temporal operator (Figure 8) that check if a formula as valid somewhere in the past.

```
assert safety_euler {
    always (all n :  Node | n in Node.visited implies once Euler = n)
}
```

Safety Property: If a node is visited, then once *Euler* was 'inside' it.

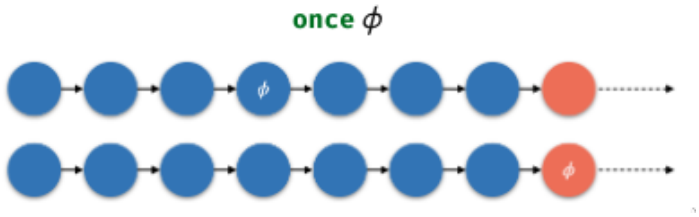

Figure 8: *once* behavioural representation.

On the other hand, a *liveness property* expresses that "something good will happen", implying the eventual occurrence of a state during the course of the system's execution [41]. This forces the system to evolve.

As a example, the following liveness property denotes the expected behaviour of the modelled example, where eventually the graph will represent an Eulerian circuit.

```
assert liveness_euler {
    eventually (adj in visited and Euler in Init)
}
```

Liveness Property: Eventually the graph will represent an Eulerian circuit.

This liveness property states the expected behaviour of an Eulerian circuit, where the *eventually* (Figure 7) operator is considered to specify the desired final state. However, its verification yields a counterexample, as it is possible to always perform the *stutter* event, and thus the expected behaviour never happens. This scenario results in an implausible infinite trace behavior, which must be excluded by adding fairness to the trace assessment.

To verify liveness properties it is advisable to consider *fairness* constraints, intentionally specified to rule out infinite traces with unrealistic behaviour, such as permanent system stuttering [4]. Briefly, a *fairness* constraint imposes fairly considerations on the system's trace evolution [66].

```
pred fairness {
  (eventually always some Euler.(adj-visited)) implies (always eventually
     traverse)
}
```

Fairness predicate: If *Euler* has an adjacent node, it will eventually visit the latter.

The latter predicate imposes a weak fairness constraint regarding the trace evolution, ensuring that permanent stuttering is not possible as long as the *Euler* node has some adjacent node that has not been visited yet. The desired liveness property is required only under fairness assumptions.

```
assert liveness_euler {
  fairness implies eventually (adj in visited and Euler in Init)
} check liveness_euler
```

Liveness property accounting the fairness constraint. It yields no counterexample.

Even though the fairness predicate denies the possibility of continuous stuttering, the liveness property still yields a counter-example, where it is depicted a graph with *2* nodes linked by an edge. The latter scenario does not describe a proper *Eulerian* graph, since the necessary condition where all nodes in the graph must have an even degree, is yet to be defined. Thus, this condition could be explicitly specified as a model constraint to duly describe a *Eulerian* graph.

2.3.2 *Alloy Analyzer*

Normally, the specification process is carried out interactively, since considerations about model validation must be preceded by the model specification. The already mentioned analysis commands instruct the Analyzer to check and provide analysis over their implicit formula nature.

Moreover, the Analyzer is capable of depicting instances graphically as graph-like structures, through the usage of the Alloy Visualizer. The latter allows instance navigation over multiple configuration buttons. Additionally, concerning the user comprehension, the graphical depiction of these instances can be customized using *Themes*, through the Theme toolbar button.

Recall the *run* example that failed to provide a model instance due to the steps scope limitation. To find an Eulerian circuit instance for n nodes, the minimum of $n + 1$ steps are required.

```
run example {
    eventually (adj in visited and Euler in Init)
} for exactly 5 Node
```

Bounded Model Checking: Eventually the graph will represent an Eulerian circuit.

Producing a concrete instance might be quite helpful throughout the modeling process. The run command presented above is capable of providing a model instance as it performs Bounded Model Checking for at most *10* steps. The Analyzer executes the command and generates an instance, consequently producing a visual graphical representation (Figure 9) through the usage of the Visualizer.

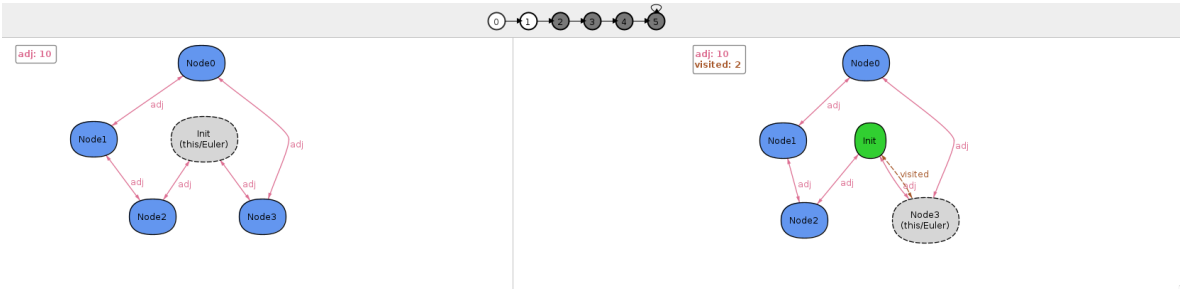

Figure 9: Partial graphical view of the *2* initial states. The *Euler* node starts in the *Init* node, and then moves towards an adjacent node.

The Visualizer has a toolbar with multiple navigation buttons, allowing interactive exploration of alternative instances [9]. Traces can also be navigated through transition buttons ($\rightarrow$ and $\leftarrow$), enabling forward and backward trace navigation.

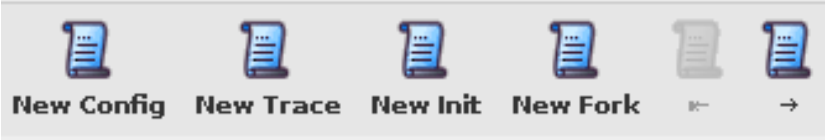

Figure 10: Alloy Visualizer toolbar.

The New Config instructs the Analyzer to provide a new trace configuration, where immutable model expressions (sets and relations) are depicted with new values. The New Trace instructs the Analyzer to present any different execution trace, regarding the same model configuration. The New Init requests for a new trace where the initial state is forced to present different model expressions values. At last, the New Fork allows alternative transition behaviour exploration over the same starting state, depicting a trace with the same prefix up to the current state but a different post-state. The latter could differ on the result of a given event, or it can display the outcome of a different event [9].

Additionally, the Alloy *Evaluator* [32] confers additional functionality to the Visualizer. It allows the user to type Alloy-based expressions against the existing model, used to gather structure information about the existing model [9].

Alloy forces the property formula translation into an LTL-based formula, considering the finitude of its universe of comprehension. Due to the emphasis on representing instances over infinite traces, every Alloy instance generated by the Analyzer, captures an infinite trace through *lassos*, where a state that loops back is reached.

Alloy considers the Bounded Model Checking technique as the default approach towards the model verification. Here, the corresponding formula is verified for all *lasso* traces of size up to a bounded number of steps. So, the verification is not complete. However, it still confers great functionality, as infinite traces can often be represented as finite traces (Figure 11), making verification within small scopes possible [35].

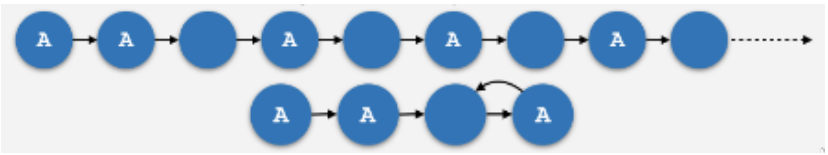

Figure 11: Some infinite traces can be represented by finite *lasso* traces.

The Analyzer can also be instructed to perform Unbounded Model Checking towards achieving consistency over model verification, without bounding traces upfront [32, 9]. This is feasible as the state space is finite due to the Alloy analysis way of bounding signatures.

To perform complete model-checking, an appropriate solver must be selected in the Options menu, and the time scope must be specified under the following syntax: $for\ 1..\ steps$. Consider the previous *run* command in terms of the Unbounded Model Checking approach.

```
run example {
    eventually (adj in visited and Euler in Init)
} for exactly 5 Node, 1.. steps
```

Unbounded Model Checking: Eventually the graph will represent an Eulerian circuit.

3

SOFTWARE DEVELOPMENT IN ROS2

Robotic systems have emerged in several scenarios, where their usage ranges between basic processes automation, up to full automation of critical tasks, consequently causing a complexity increase in this domain.

Due to the wide variety of robotic hardware existent in multiple domains, robotic software development is rather difficult [18]. The reuse of code is non-trivial, and therefore, large-scale development can become rather untenable. The *Robotic Operating System* (ROS) presents itself as a middleware system, created to facilitate robotic system development in large scale.

In ROS, software flexibility was prioritized above all else, meaning that values like security were disregarded. Thus, ROS-based applications tend to face increased security risks, related to the exposure of the whole robotic network. Due to the scale and scope of the robotics growth, security guarantees must be addressed as a developing priority [26, 38].

The new version of ROS, *Robot Operating System 2* (ROS2), presents itself as a framework for developing robotic systems supported by the *Data Distribution Service* (DDS) standard. Multiple implementations exist for this standard, which implement numerous DDS-based specifications as well as valuable *Quality of Service* (QoS) transport parameters.

The *DDS-Security* specification[1] aims to supply multiple plugins regarding the security domain. Consequently, ROS2 yields a wider command toolset compared to the former version of ROS, as they bring forth to a toolset, the *Secure Robot Operating System 2* (SROS2) toolset, concerning the security functionality that DDS-Security plugins offer.

This chapter introduces necessary background information about the major concepts required by this dissertation. First, it presents a detailed introduction to the concepts around Robot Operating System (ROS), as well as the evolution approach that ROS faced towards providing security to its deployed systems. Here, the Data Distribution Service (DDS) and its integration in ROS2 must be contextualized beforehand.

1 https://www.omg.org/spec/DDS-SECURITY/1.1/

3.1 ARCHITECTURE CONSIDERATIONS

The Robot Operating System was created by a collaborative open-source community, that has undergone rapid development [18] to contribute to the advancement of cyber physical systems. It was purposefully designed to be a development enhancer for the realm of robotic applications [26, 55].

Fundamentally, ROS is a middleware, providing a custom serialization format, a custom transport protocol, as well as a custom central discovery mechanism, presenting itself as a distributed layer between the top application layer and the operating system layer.

ROS was designed to provide as much as modularity and composability to the application layer as possible [12], allowing ROS applications to be built over several software modules, as independent computing processes called *nodes*. These are composed together to fulfill the deployment goal of the corresponding robot [48].

3.1.1 *Former Architecture*

The Robot Operating System architecture is based on a hybrid peer-to-peer implementation, where network communication is done over message-passing through a publish-subscribe pattern. The communication API relied on a stateless XML-encoded remote procedure protocol. Transport libraries provided data exchange accounting serialization over sockets [69, 25].

This architecture (Figure 12) emphasized on approaching communication through a centralization perspective. It relied on the explicit implementation of a *Master node*, that controlled every aspect of the communication establishment. Thus, the initial information exchange within the network had to go through this master, that is then converted in a peer-to-peer communication.

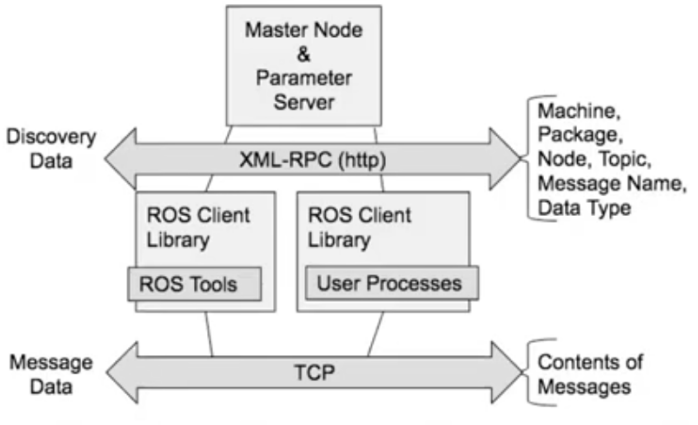

Figure 12: Robot Operating System architecture.

Formerly, due to the sheer wide capabilities controlled by the master, this centralization approach was duly valorized. It naturally fits the purposes of a research tool, as it is simpler to monitor and analyze the system behaviour. However, because it is strongly reliant on the master node's availability, this communication architecture

does not scale effectively, making it unsuitable for safety-critical or real-time applications. If the master fails, the entire system fails, representing a single point of failure and a huge performance bottleneck.

Many research communities tried to fix these real-time issues by proposing potential solutions, while supporting the same architecture design. Unfortunately, they fell short of meeting the requirements of real-time applications. It became clear to the ROS community that the framework had architectural limitations that could not be rearranged using the same design approach [48, 25].

The Robot Operating System 2 comes as a complete refactoring of ROS, with the aim of increasing the framework's real-time capabilities, by allowing the development of time-critical control over ROS. It moved away from the former architectural design towards the use of an external communication middleware that can support the production needs of the outgrowing robotic systems [38, 12].

3.1.2 *Data Distribution Service*

The Data Distributed System (DDS)[2] is an *Object Management Group* (OMG) middleware standard. The standard was developed to address the demand for enhanced interoperability across different vendors' frameworks, directly addressing data communication between nodes that belong to a *publish-subscribe* communication architecture, and particularly targeted for real-time and embedded systems.

A communication middleware aims to ease the complexity behind creating and maintaining communication architectures. It is responsible for handling relevant aspects like network configuration, communication establishment, data sharing and low-level details. As a result, system developers can mainly focus on their applications purposes, rather than concerning about information moving across levels [29].

DDS architecture (Figure 13) uses the *Data-Centric Publish Subscribe* (DCPS) model as its communication model approach. DCPS is based on a publish-subscribe pattern, where the *data-centric* messaging technique is implemented. It conceptually creates a virtual *Global Data Space*, acessible by any DDS-based application, where data is properly delivered to the applications which request for it, saving bandwith and processing power [52, 54]. A *domain participant* enables an application to participate in the Global Data Space, either as a *publisher* or as a *subscriber*, according to their role on data exchange [48, 1, 20].

To properly address the data transportation through the physical network, DDS offers a wire specification protocol called *Real-Time Publish-Subscribe Wire Protocol* (RTPS) [62], providing automatic discovery between participants. This protocol also works under a publish-subscribe policy over best-effort transports, where data transmission between endpoints is handled [74]. RTPS allows multiple applications, that could differ on their used DDS implementations, to interoperate with each other as network domain participants [20, 1].

Furthermore, RTPS was designed to employ Quality of Service (QoS) profiles, which allow for the specification of various transport policies, formerly not covered by DDS. This approach offers flexibility over communication configuration and development versatility, allowing the developer to specify whatever QoS satisfies its system's communication needs [1, 26, 48].

2 https://www.omg.org/omg-dds-portal/

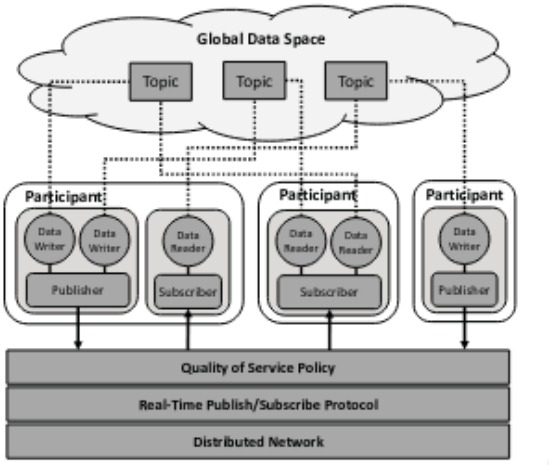

Figure 13: DDS architecture: DCPS model with RTPS. Extracted from [48].

Briefly, DDS leverages the premise of a transport-independent virtualized *Data Bus* (Figure 14) to address network resources' distribution, in which stateful data is distributed through the network. The involved applications can access this data in motion, representing an architecture with no single point of failure, respectively enabling a realiable way of ensuring data integrity. Consequently, by adopting this approach, the load on the network is independent of the number of applications, making it easily scalable.

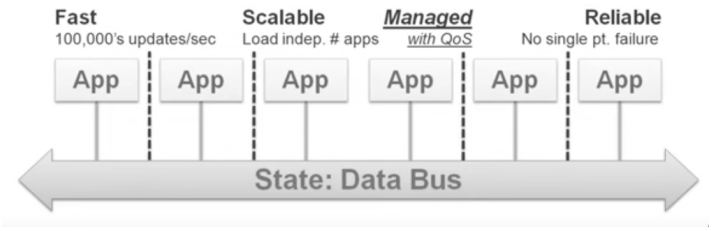

Figure 14: Data Distributed System communication in a nutshell. Extracted from [29].

3.1.3 *ROS2-DDS Architecture*

As previously stated, the Robot Operating System 2 was developed to address the former architecture lack of support for real-time systems, mainly due to its architecture design that relied on their own middleware specification. To address this, ROS2 middleware architectural approach (Figure 15) is built upon the DDS framework [48], leveraging DDS for its messaging architecture, where communication and transport configuration are handled.

As far as dependencies are concerned, DDS implementations have light dependencies, often related to language implementation libraries, easing the complexity behind installing and running dependencies [72].

The middleware's top layer regards the ROS client library (*rcl*)[3], already implemented in the former ROS architecture. This layer accounts the availability of ROS concepts to the Application layer, as it provides APIs

3 http://wiki.ros.org/Client%20Libraries

to ease the software implementation by ROS developers [57]. As ROS aims to support different programming languages over the same computing context, each language-specific API must have its corresponding client library (*rclcpp* regarding *C++* and *rclpy* regarding *Python*). The *rcl* hides these client libraries by abstracting their specification, consequently reducing code duplication [63, 12].

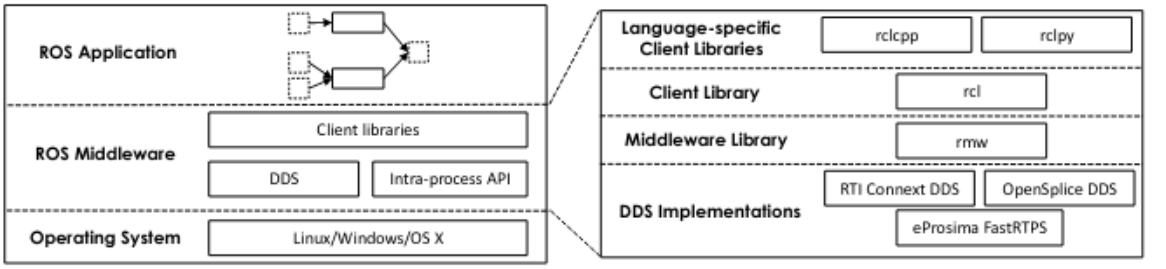

Figure 15: ROS2 framework architecture. Extracted from [12].

With the goal of supplying a wide range of configurations back to application layer, ROS2 supports multiple DDS implementations, in which these implementations API specification differs from each other (currently, ROS2 supports *FastRTPS* by *eProsima*, *Connext* by *RTI*, and *Vortex OpenSplice* by *Adlink*). It should be noted that the DDS implementations are at a low-level of abstraction, strictly defined by its corresponding vendor's API. DDS only defines fundamental procedures at a higher degree of abstraction.

In order to abstract *rcl* from these implementations APIs, an DDS-agnostic interface was introduced, the *rmw* (ROS MiddleWare) interface [12], allowing portability among DDS vendors, which consequently enabling ROS developers to use several DDS implementations, based on their applications needs during runtime. The information flow through the middleware layer is done over structures mapping between ROS and DDS data models, addressed by the *rmw*. Here, the DDS implementation that is being considered at runtime is addressed by the *rmw*.

3.1.4 *Computation Graph*

From a logical perspective [12], ROS applications are composed of many software modules that operate as computation nodes, allowing its participation into the ROS Global Data Space. The use of the publish-subscribe as communication paradigm, through message-passing patterns, confers additional complexity to the application architecture, where the latter can be naturally represented as a computation graph [17].

The application's computation graph presents itself as a graphical network, where runtime named entities have their unique role when it comes to data distribution.

Node Instances

The application development is done via package orchestrating, where each logically represents a useful software module. Packages might be comprised by numerous *nodes*, that can be perceived as processes that will

likely perform computation over the network. It is worth mentioning that, nodes can be connected within a single package or between multiple packages [17, 55].

Thus, the network is comprised by many nodes, running simultaneously and exchanging data between them, where each node addresses its corresponding purpose [57]. Fault tolerance features are guaranteed as nodes have their corresponding unique name, allowing communication in an unambiguous manner, which confers a suitable approach to develop a complex robotic system.

The notable usage of callback functions provides great functionality when it comes to manage node's behaviour in the communication process. Additionally, *timers* can also be used, since they provide a useful way of managing these callbacks, by time-assigning.

Communication

Message-passing is the primary means by which nodes communicate with one another. The *message* definition is a well-typed data structure, which commonly characterizes every data structure concerning the information exchange between nodes. A message is defined by its data type, also known as its *interface*, which can either be primitive (*integer*, *string*, *boolean*, among others), or defined by a complex data structure, where multiple data types are assigned to their corresponding fields [57, 55].

The ROS computation graph provides *3* different ways of establishing node communication, those being *Topics*, *Actions*, and *Services*. These communication mechanisms have different interfaces, specified in different folders with unique namespaces [57].

Topics are perhaps the most common method, naturally perceived as middle-communication buses, over which messages are passed through. Communication through topics is handled by the publishing-subscribing pattern (Figure 16). A node publishes the message to any number of topics, that are then subscribed by nodes that want to get access to that message. Topics provide a multicast routing scheme, where publish data is casted into the multiple nodes that are subscribed to the topic [12].

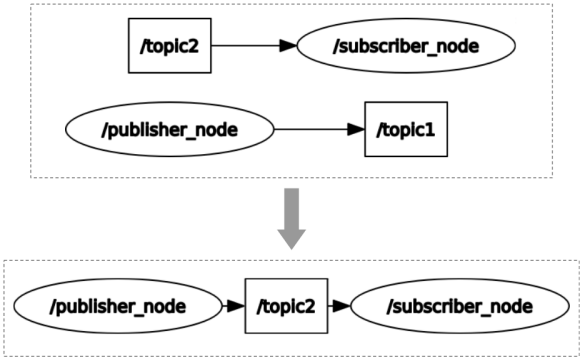

Figure 16: Communication behaviour over *topics*.

A specific *topic* is created upon specifying its entity name in either a publisher or a subscriber callback instance. Whenever a node creates a publisher, with the intention of publishing messages through a specified topic, *roscore* is used to advertise the latter, enabling message passing to the corresponding topic subscribers. Message

processing is done via the node's callback functions, which are activated upon message reception, and which in turn can be used for publishing further messages [12].

Even though topics are the most conventional way of communication, due to its multicast scheme, subscribers can not be identified by the publishers, so logging and synchronization becomes rather difficult [55].

The use of services enables a client node, that can also be seen as a topic subscriber, to request data from a server, that likewise a topic publisher, provides data through a service. This is a bidirectional synchronous form of communication based on a request-response pattern.

Other notable way of exchanging data is by setting goals through Actions. Actions are intended to process long-running tasks, where the client sends a goal request to the server node, that confirms the reception of this goal. The server might provide feedback to the client before providing a final response.

Launch Files

A conventional way of deploying a ROS application is through the use of launch files, enabling the multi-configuration over entire robotic applications, where network nodes can be individually pre-configurated. ROS makes use of the *roslaunch* to automatically initialize the whole network, simultaneously launching each node [55]. This provides a simpler way of monitoring the system nodes.

Figure 17 depicts the network architecture corresponding to an ROS application based on the well-known *TurtleSim*. This application is mainly composed by *two* nodes, that work together towards moving a turtle. Additional nodes were implemented, in order to add complexity to the current network, to later illustrate the need for security guarantees.

Briefly, the *multiplexer* node acts as a topic selector between two different subscribed topics, where each of them was respectively associated with a priority value. The remaining nodes implemented, namely the *keyboard* and *random*, publish to one of the *multiplexer*'s subscribed topics. Notably, the *keyboard* node has priority over the *random* one, meaning that when the *keyboard* publishes movement data through its corresponding topic, the *multiplexer* must instantly process the received data. Based on the priority valued, the *multiplexer* node forwards the commands, related to the selected topic, into the *turtlesim* node, triggering the turtle's movement.

The *rqt*[4] visualizer, *rqt_graph*, allows the developer to perform analysis over a graphical visualization of the network computation graph.

Figure 17: *TurtleSim*'s network graph presented by *rqt_graph*.

After the proper configuration of each node, the network can be easily managed and automatically launched through a launch file. Figure 18 shows the launch specification related to this application. Note that, each *node*

4 ROS provides a GUI tool called *rqt*, that assists developers in manipulating the network elements, in a more user-friendly manner.

tag points to a specific package (*pkg*) and executable (*exec*). Here, each executable might share the same package, where for instance, the *keyboard* and the *turtlesim* share the same package. The *name* field enables the specification of distinct namespaces to each executable.

```
<launch>
    <node name="turtlesim" pkg="default" exec="turtlesim" output="screen"/>
    <node name="keyboard" pkg="default" exec="keyboard"/>
    <node name="random" pkg="random" exec="random"/>
    <node name="multiplexer" pkg="multiplexer" exec="multiplexer"/>
</launch>
```

Figure 18: *TurtleSim* launch file.

Additional node configuration, such as name remapping and parameter adjustments, can be specified under the *args* tag, which offers great functionality and flexibility to the launching process. Remapping represents a powerful ROS feature, allowing multiple configuration under the same network launch file. Node remapping provides for the reuse of the same node executable in other areas of the system, whereas topic remapping allows for behavioral mapping across distinct topics [57].

Distinctive namespaces allow the system to start the nodes without any name nor topic name conflicts. However, this technique has some flaws, namely it does not furnish a way of launching nodes in a separated terminal, often needed for user interaction purposes, like input reading.

Parameters

Another relevant concept behind ROS is the existence of node parameters, that can be used, for instance, for the configuration of the network nodes. In the former version of ROS, the node parameters were controlled by a global *Parameter Server*, managed by the corresponding ROS Master [55]. However, in ROS2 each node declares and manages its own parameters, by using the predefined commands *get* and *set*. Additionally, using a callback, the node's parameters can easily be edited [57].

Node Composition

Usually a node is attached to a single process, but it is possible to combine multiple nodes into a single process, structurally abstracting some network parts, while improving the network's performance [57]. However, there is a slight difference about how ROS and ROS2 approach node composition.

In the former version of ROS, node composition was done over the combination of *nodelets*, intentionally designed to ease the cost of overusing TCP for message-passing between nodes. Here, nodelets enables publish and subscribe calls through a *C++* boost pointer, without incurring memory over-utilization and copy when passing messages within the same node. This technique eliminates message forwarding inefficiencies while preserving the modular nature of ROS nodes [46, 57].

Inspired by the former idea of nodelets, ROS2 introduces the *components* as software code compiled into shared libraries, that can be loaded into a *component container* process at runtime in the network, ensuring node composition [57].

3.2 SECURITY

The deployment of real-time systems implies a special care about safety and security [48], resulting of the demanding time-critical scenarios. Robotic systems fall under the umbrella of this broad system definition, as they feature unique cyber vulnerabilities related to its integration over highly networked environments, that confers great importance on exposing critical time-reliant scenarios [49, 24].

3.2.1 *ROS Security Problems*

The network security evaluation in a system is done by applying several analyzing techniques. Generally, these techniques do not cover every security aspect, as new vulnerabilities arise from technology evolution [37]. The application of security countermeasures techniques upon configuring the system's network confers a critical step when trying to achieve security. Network security entails pre-exploration of the system's network through practical networking security techniques, such as intrusion detection and traffic analysis [47].

Numerous researchers [21, 25] have investigated the use of these techniques, such as port scanning and penetration testing, over the previous version of ROS in order to thoroughly assess attack vulnerability throughout the ROS architecture.

The ROS Master role in the communication architecture, and its ability to connect to other nodes, imposes many concerns about how to address security, namely to ensure protection of the Master node. Exposing this node poses a critical threat for the whole network [21].

Moreover, there were also worries regarding the way ROS handled node communication. Network security may be jeopardized, as a result of the publish-subscribe pattern transparency, where node-to-node communications are done in plain text, making data content vulnerable to unauthorized usage [38, 69]. Additionally, the former API did not confer any message integrity technique, making applications vulnerable to packet sniffing and man-in-the-middle attacks [69].

However, due to the high non-linearity and complexity of real-time systems, implementing such thorough analysis method in near real-time remains a significant difficult task [22].

3.2.2 *DDS-Security Specification*

The Object Management Group accounts security integration by supplying an in-depth specification, consequently adding features to the previously developed DDS standard. The DDS-Security is a specification that serves as a security extension to the DDS protocol, defined by a set of plugins (Authentication, Access Control,

Cryptographic, Logging, Data Tagging), combined in a *Service Plugin Interface (SPI)* architecture (Figure 19) [5, 28].

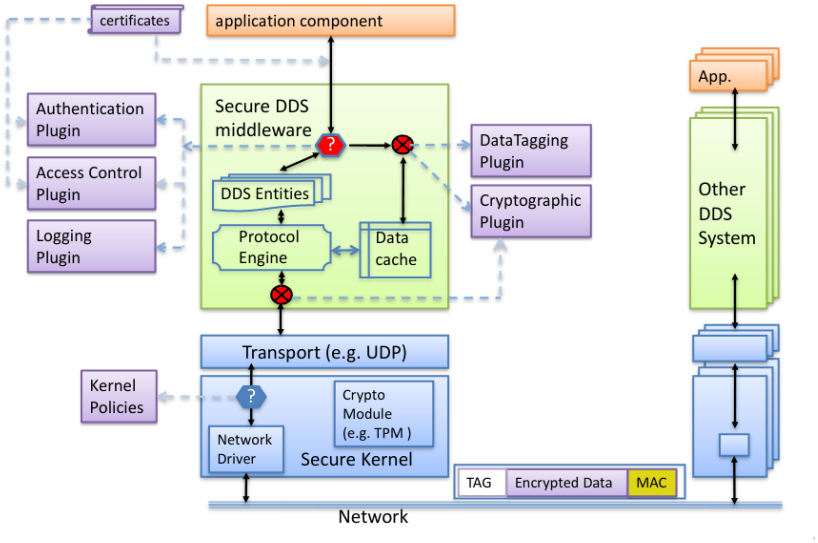

Figure 19: DDS-Security Architecture. Extracted from [53].

This specification enables its integration by furnishing a *Security Model* supplied to the DDS standard, whereas the *Service Plugin Interface* architecture is responsible for granting plugin enhancement for compliant DDS implementations. Moreover, depending on the security requirements needed for a particular application, these plugins might be adjusted by the latter's runtime DDS implementation [53].

Authentication

Upon considering a secure environment over the DDS Global Data Space, data integrity can not be prone to unauthorized usage. Therefore, data exchange requires verification procedures to properly identity the authenticity of each DDS domain participant.

The Authentication plugin is the most important plugin of the entire SPI architecture, as it provides means to validate the identity of each application, later regarded as a domain participants [53, 28].

Each participant must be authenticated prior to entering the data space [71]. To implement such secure environment, participants are presented to the secured environment regarding the *Public Key Infrastructure* (PKI). This is in charge of issuing asymmetric keys to each participant, a *public* and *private* key respectively, that are used for both authentication and encryption procedures [26, 28].

Moreover, regarding the issuer identity, the PKI provides an *X.509 certificate*, that maps a distinguished domain name (DN) to the participant's public key [26, 69]. This certificate is accountable and signed by a trusted certificate authority (CA) [71, 69, 28]. It ensures authenticity between asymmetric key pairs (both public and private) [26].

The communication establishment over different participants must be preceded by a mutual handshake, where keys and certificates are exchanged to guarantee their authenticity [71, 38]. Additionally, the DDS permissions of

a domain peer are also considered in this handshake. The control over permission distribution is respectively handled by the Access Control plugin.

Access Control

As aforementioned, the defined DDS specification handles policy control over the DDS domain through the Access Control plugin, where authenticated parties operations are controlled by policy restrictions [53, 71]. Domains within the Global Data Space are controlled over a set of DDS-related capabilities, that are either assigned or restricted to the authenticated participants [28].

Authenticated participants must be granted access to certain domains, where their roles on data transportation must be accordingly accounted by access permissions. If a participant is perceived as a domain data publisher, the domain restrictions must provide publishing privileges to its data topic [71].

Following the authentication procedure, domain authorization is also implemented using the PKI [28], by embedding policy definitions through certificate extensions [69].

Furthermore, the Access Control plugin employs two configuration documents that are allocated to each participant [71]. This provides significant security capability, which is given as a supplement to the authentication procedure.

- Domain Governance: *XML* document defining the domain's security policy.
- Participant Permissions: *XML* document containing the permissions assigned to a given domain participant.

Notably, these configuration files are signed by a trusting Certificate Authority (CA) [28]. The CA's Permission Certificate confers protection against elevation of privilege attacks. Therefore, if the policy integrity is jeopardized, the handshake establishment between authenticated parties fails to succeed [69].

Communication and Encryption

The DDS-Security specification ensures encryption and authentication using *OpenSSL*[64]. Accordingly, the OpenSSL protocols, *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS), are respectively used to ensure encryption over the network communication [69, 38]. The handshake is used to achieve mutual authentication within participants over the DDS domain [71].

Following the participant key assignment, the *Diffie-Hellman* key exchange protocol properly accounts the mentioned handshake, allowing both participants to exchange data over a shared secret key [38]. It enjoys the exchange of both asymmetric keys, issued to each domain participant during the domain authentication procedure, in order to perform sign (using the private key) and verify (using the public key) operations [38, 26].

The *rcl* is capable of handling this DDS security requirement, levering ROS-based applications to support SROS2, accounting nodes as authenticated participants [69].

As communication establishment is duly achieved throughout this handshake process, the DDS-Security specification takes advantage of the *AES-GCM* encryption standard to ensure data encryption over the channel [38, 64].

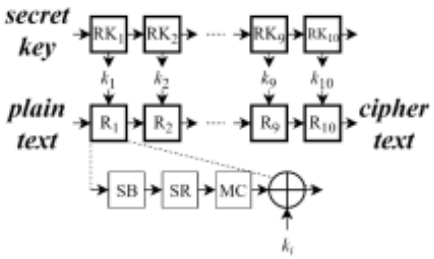

Figure 20: *Advanced Encryption Standard* algorithm. Extracted from [64].

The key cryptosystem algorithm (Figure 20) presented in the *Advanced Encryption Standard* (AES) considers the usage of functions towards achieving data encryption over the established communication. It enables configuration over data encryption, allowing both block and streaming transfer of ciphered data [26]. Here, the algorithm combines the shared key established with the message passed over the secure channel. Moreover, a *Message Authentication Code (MAC)* encryption function is added to the AES algorithm for increasing encryption functionality [64, 38].

Every cryptography-related operation is handled by the SPI's Cryptographic plugin. Naturally, these cryptographic capabilities, linked to the asymmetric key cryptography, are then used by both Authentication and Access Control plugins, ensuring authenticity support over their corresponding opearations [28, 26].

3.2.3 *Security Integration in ROS2*

As result of the DDS implementation as a flexible middleware interface in the ROS2 architecture, issues regarding security are no longer mainly ROS-dependent. Thus, when it comes to addressing security over communication, and subsequently data protection enhancement, ROS2 is heavily reliant on how the DDS standard is able to manage security [38, 21].

Every DDS implementation supported by ROS2 makes use of the DDS-Security specification, enabling security over ROS's application environment. Even though ROS2 is deployed without security mechanisms by default [28], ROS2 provides a toolset, the Secure Robot Operating System 2 (SROS2) toolset[5], extending ROS2's functionality to make use of the DDS-Security functionality.

The control over these tools is done by *rcl*, providing security to the Application layer, while DDS is capable of providing security of the communication architecture [38]. SROS2 configuration is done by defining a set of security files to each ROS2 participant, with regard to how DDS handles certificate assignment [69, 28].

This toolset allows certificate generation and allocation through a keyserver directory, where security enclaves and their respective authentication files are stored inside subfolders [69, 28]. Moreover, SROS2 enables the keyserver configuration in a flexible way, allowing developers to restricit certificate and CA attributes to an arbitrary number of nodes, defined over the same namespace [69].

5 https://github.com/ros2/sros2

The variety of capabilities in SROS2 toolset attempts to aid with security configuration across environments [28]. However, managing certificates and access control policies might lead to improper configuration. Additionally, orchestrating a real-time network towards achieving a secure environment confers to be a demanding task [28, 71].

Security Enclaves

As aforementioned, ROS2 relies on how handles DDS security over their *Domain Participants*. DDS imposes the authentication of each participant prior to joining its Global Data Space, where the Certificate Authority and an established PKI comes in hand [71, 69].

Accordingly, the authentication process within the ROS2 network relies on the notion of a network enclave, where authentication and control artifacts are stored to properly achieve security over the network data space [67]. Conceptually, an enclave is a secure region in the application address space that maintains confidentiality and integrity, while computations are being carried out on data.

Previously, a node was perceived as a separated DDS participant. However, by considering node composition as a reliable way of matching multiple nodes simultaneously to the same security domain, this node perception as participants can not be taken into account, due to causing non-negligible overhead of combining different participants per process. Additionally, each participant's memory space becomes rather difficult to handle [67, 68].

Concerning the enclave authentication procedure, its security artifacts must be addressable by a DDS participant, where the latter is associated to a node sharing context, instead of being perceived as a single node [67]. Thus, when it comes to apply different policies over different nodes, that are matched in the same node context, a set of node profiles can be specified under the enclave notation.

ACCESS CONTROL WITHIN ENCLAVES The SROS2 toolset offers a well-typed markup language *XML schema* [68], where security policies bind profiles to access permissions for network objects, granting privileges back to a certain profile. *Profiles* are implemented under the *enclave* declaration, to duly support the node composition into a single process, enabling the possibility of combining multiple profiles, respectively addressing its corresponding node.

Objects are classified over a subsystem type, structurally characterized by permissions tags. Then object privileges are controlled with access values, either *allow* or *deny*, attributed to their corresponding permissions tags [68, 70]. Briefly, each node profile is assigned to policies that concern some object identifier (OID). Each OID maps to a specific action, that is identified with an access value, allow or deny respectively [69].

The policy design approach works under the *Mandatory Access Control*, that denies any privilege by default. The only way of allowing access to any object, is by explicitly specifying the subject's privilege access [68, 70, 69]. Naturally, this policy approach confers great value towards security, since it denies any attempt of privilege gaining attack by ROS-based packages from non-trusted sources [69].

Figure 21 depicts a policy file where access control privileges are distributed across enclaves, and their inherited profiles. This policy file is for the *TurtleSim* example and it addresses the topic's privileges of each respective enclave. Since each node within the network has distinct privileges to the network topics, the considered policy file regards each node with a corresponding enclave.

```
<?xml version="1.0" encoding="UTF-8"?>
<policy version="0.2.0" xmlns:xi="http://www.w3.org/2001/XInclude">
  <enclaves>
    <enclave path="/multiplexer">
      <profiles>
        <profile ns="/" node="multiplexer">
          <topics publish="ALLOW" >
            <topic>move_turtle</topic>
          </topics>
          <topics subscribe="ALLOW" >
            <topic>high_priority</topic>
            <topic>low_priority</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
    <enclave path="/turtlesim">
      <profiles>
        <profile ns="/" node="turtlesim">
          <topics subscribe="ALLOW" >
            <topic>move_turtle</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
    <enclave path="/keyboard">
      <profiles>
        <profile ns="/" node="keyboard">
          <topics publish="ALLOW" >
            <topic>high_priority</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
    <enclave path="/random">
      <profiles>
        <profile ns="/" node="random">
          <topics publish="ALLOW" >
            <topic>low_priority</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

Figure 21: SROS2 policy file regarding the access control policies of the *TurtleSim* example.

3.3 RELATED WORK

As reliance on robotic systems grows due to their expanding application across a wide range of domains, these systems concern critical scenarios, where human interaction comes in hand [26]. Thus, safety should be highly considered when employing systems that might jeopardize the human's integrity, such as robotics. Additionally, analysis regarding the system's quality assurance is increasingly becoming a focus of attention. Consequently, it is critical to employ techniques that promote the increase of the system quality, without sacrificing its flexible nature.

Even though this dissertation is devoted to a detailed examination of ROS framework regarding its security deployment, there are many similarities to other popular multi-configurable robotic software in terms of quality assurance and security overview. As result, a broader range of robotic systems will be reviewed in this section to give contextualization of some relevant aspects that also fall under the ROS domain.

The subsection 3.3.1 aims to provide a comprehensive overview over works that attended to prevent security issues related to the ROS architecture. It is then followed by the subsection 3.3.2 that concerns about previous work addressing property verification and model checking techniques over the robotics field of exploration.

3.3.1 *Security in ROS*

This section confers a notable overview of ROS security. It begins by presenting some works that demonstrate the absence of security of the ROS environment, followed by some attempts to address this problem. Finally, ROS2-based studies will be given to contextualize the dissertation's main topic of study.

Robotic systems were initially conceived as augmented computers with no explicit boundaries or limitations. As a result of the requirement to provide practical systems as fast as possible, security matters were disregard [70]. Network security entails cautious analysis of the system's network using realistic networking security procedures [47]. Concerning the Robot Operating System and its role as a robotic application enhancer, numerous researchers have examined the usage of such procedures to perform a thorough analysis over the latter's architecture.

DeMarinis, Tellex, Kemerlis, Konidaris, and Fonseca present a practical overview of ROS security, in which the *IPv4 address space of the Internet* is explored with the goal of identifying vulnerable hosts. Port scanning was used as technique to expose mostly master nodes as they provide valuable information about their associated topics and node's parameters. The performed scans gathered information about hosts that could either be a sensor, an actuator or even a simulator. This study is rather relevant because it shows how easily can attackers gather information about existing robots, and control them remotely, through the public Internet, making it unavoidable to develop mechanisms concerning security.

Moreover, in [25], different pentesting tools that entails exploiting techniques over ROS-based systems, in order to provide a proper overview of possible security flaws. Foremost, Dieber et al. present a *.net-based pentesting* tool called *ROSPenTo*, developed with the intention of investigating strategies for manipulating running ROS applications. The latter provides several command line techniques to jeopardize robotic networks, including the ROS parameter server.

The same paper presents another tool called *ROSchaos* wittingly designed for exploiting the Master API. It differs from the former tool, since *ROSPenTo* mainly focused on exploiting ROS *Slave API*, which covers the node-to-node communication and the reception of messages from the Master, without directly addressing the ROS Master as a compelling target. Regardless matter how subtle such attacks are, exploiting the Master directly may still be appealing to attackers.

These techniques show the relevance of security in ROS, in particular how the *XML-RPC* embedded API can be breached and subsequently exploited according to the participants roles within the network. Besides raising awareness of the importance of security in ROS, it alerts developers for the need to conduct penetration testing on their applications.

Following the challenges that arose as a result of executing exploitation techniques on the ROS framework, numerous solutions were proposed in response to the need to offer security assurance for robotic applications.

One of the earliest research on the security improvement of the ROS framework was proposed by McClean et al. in [49]. The first take is to exploit and reason about unique vulnerabilities related to the nature of cyber-physical systems. Afterwards, it presents a novel research tool to aid in cyber-physical security research, named as *honeypot*, that monitors unauthorized attempts to jeopardize these systems. The latter gives a basic yet crucial study on ROS, which acts as assistance to steer the development of future work in the burgeoning subject of cyber-physical security.

In [69], White et al. address the security deployment over ROS in a more concise way, by proposing the *Secure Robot Operating System* (SROS) as an enhancement to the former API, that includes mechanisms such as authentication, access control and cryptography measures. SROS seeks to provide an adequate security architecture while minimizing disturbances such as computational cost and API breaking. The final remarks regard the alternative usage of SROS, allowing developers to customize it to their own internal certificate formats. Moreover, White et al. expect security logging and access control to evolve through well-defined standards, enabling more robust auditing and policy generating tools.

Additionally, in [23] it is presented a fairly pertinent research that recommends security improvements on the application-layer. Briefly speaking, the approach primarily focused on applying security measures on the application layer, by mainly running an Authentication Server, storing certificates and files related to trusted domain participants, while controlling and providing session keys related to the communication process. Topic-specific encryption keys are employed to protect data secrecy. However, the discussed architecture is based on the assumption that authentication certificates are manually issued. Despite encryption and authentication mechanisms, attacks regarding the exposure of the network, such as denial of service attacks, still persist which cannot be handled on the application level alone.

Breiling, Dieber, and Schartner continued to follow the previous work in [7], through a hardened ROS core with transparent authentication, authorization, and encryption functionalities that do not require the manual specification of nodes. Furthermore, secure workflows and initial penetration testing are supported in [24], in which Dieber et al. show an insight of security weaknesses in the ROS architecture design, with additional attempts that regard potential solutions to those problems.

Regarding the SROS [69] initial proposal, White and colleagues continued to provide security mechanisms to the framework in subsequent studies [70, 71]. In [10], Caiazza, White, and Cortesi present several solutions regarding the lack of security measures that autonomous devices, often related to the robotics world, tend to face. The effort then moved on to evaluate the ROS framework in order to provide a high-quality solution. It follows a pipeline of security measurements, where logging, access control and authentication certificates are discussed over several proposals. It provides a clear analysis over the security state-of-art of ROS, and the analysis clearly states that the lack of techniques to prevent communication threats imposes the most valued consideration over robotic networks. Caiazza, White, and Cortesi conclude their work by stating some future improvements to their current solution.

In [70], White, Christensen, Caiazza, and Cortesi address robotics security through a proposed framework that focuses on handling access control policies for robotic software, with the intention of adding functionality to SROS [69]. The latter offers an interesting perspective on leveraging access-control through a well-typed markup language schema, the *ComArmor* configuration language, where privileges of objects are explicitly defined over policies. Then, profiles are used to arrange these policies in a hierarchical manner, binding namespaces to objects privileges, using attachment expressions with predefined permissions denoted as rules. Rules are defined as a set of permissions for a given object, where each permission has a corresponding value that could either be allow or deny.

The evaluation of this policy schema as a proof of concept was done by implementing it in a cryptographic framework called *Keymint*. The latter follows similar authentication and cryptographic patterns as the initial SROS [69], where security artifacts and document keys are stored in respective keystores. The incorporation of the ComArmor language into Keymint adds significant value to the framework since it allows for reasoning about how to manage policy control prior to implementing any authentication mechanism, such as generating permission and governance files. The framework is then tested using a standard ROS example of a basic publisher and simple subscriber interacting over a topic. Thus, this approach provides a systematic testing procedure where both application and access control policies implementation are overviewed, adding significant value back to the framework as it is possible to define higher level policies to allow for transport-independent access control definitions.

The ComArmor configuration language, presented above in [70], is an alternative to the SROS2 default policy schema. During the framework's evaluation process, White et al. present a set of gratuitous permissions within SROS2 template through a simple comparative study between both configuration languages. Thus, it discusses possible SROS2 vulnerabilities discovered during the development and experimentation of the Keymint framework, emphasizing the importance of continuous security evaluation throughout the design development cycle.

Following the deployment of DDS in ROS2, the majority of security problems were alleviated. As a result, the literature on the network security enhancements provided by ROS2 is largely concerned with providing an overview of the trade-off between performance and security. Furthermore, various studies evaluate ROS2 performance in terms of response time over safety-reliant circumstances [48, 12], in which DDS capabilities are highly emphasized to properly evaluate ROS2.

A notable work is presented in [38], where the security definition of DDS was explored using current security standards. Also, the implementation of the DDS protocol was evaluated using techniques of static analysis. It provides a framework for doing benchmark evaluations in various network and security configurations to reflect on the needs of real-time applications. Different performance time-related metrics and benchmark network scenarios (wired and wireless) were thoroughly investigated to give a well-based analysis over efficiency. The findings clearly illustrate that employing a reliable VPN application is more time efficient; nevertheless, SROS offers far more decentralized features than a VPN, which is typically desirable when designing a distributed system. Kim, Smereka, Cheung, Nepal, and Grobler conclude their work by reasoning about the security capabilities that DDS has to offer, after which a static code analysis of the DDS security implementation was undertaken. The research revealed that the latter did not conform to the security specification by OMG.

A pertinent research that examines the trade-off between performance and security is presented in *Robot Operating System 2: The need for a holistic security approach to robotic architectures* [26]. Here, DiLuoffo et al. conduct a thorough examination of ROS2 and outlines possible hazards for this new robotic system paradigm. Thus, the DDS security specification is thoroughly examined in terms of performance versus security models and how security is integrated with ROS2, as well as how the addition of security affects system design.

As a result, DiLuoffo et al. go on to present a systematic study of ROS2 in order to provide a good knowledge of how the system is built, and to subsequently undertake vulnerability analysis across various layers that require varying levels of protection. Additionally, the authors applied different performance metrics to duly present a well-based comparison in whether security should be considered or not. This experiment illustrates that implementing security measures on protected data in motion results in a significant performance loss.

Notably, they conclude with some recommendations towards security areas not duly covered by the DDS specification, such as attacks stemming from spy processes. They highlighted the preservation of the cognitive layer because of its importance in the robot itself, since all types of sensors that involve data collecting are part of that layer. Here, an advanced set of threats were described using Machine Learning, which DDS is not capable of mitigating.

These presented works denote important considerations about how important is to address security in robotics, due to the variety of attacks that might compromise their integrity. They follow the path of addressing vulnerabilities through applying security methods based on tools and protocols. However, studies regarding the appliance of quality analysis over static and property verification in ROS are quite limited. Despite this, Section 3.3.2 will describe several works that advocate using formal quality analysis to assess robotic systems.

3.3.2 *Verification of Robotic Systems*

Analysis through property verification in the ROS framework represents a major contribution to this dissertation, namely works that aim to tackle issues arising from miss configurations or code inconsistencies. Although researches and tools on analysis of ROS systems are scarce, there are some works that might be useful within the context of this dissertation.

Cortesi et al. in *Static Analysis Techniques for Robotics Software Verification* [16] give a proper understanding on how important it is to perform formal analysis to ensure the reliability of the resulting software, while significantly minimizing the testing effort. In this work, the authors intend to cover multiple static analysis techniques to properly contextualize robotic developers about which is the most appropriate analysis strategy to follow based on the kind of examined property and software system. After briefly introducing each technique, the authors present an overall evaluation of the various static analyzes techniques, following several criteria: automation, precision, scalability, and soundness. Cortesi, Ferrara, and Chaki conclude stating that performing analysis techniques is highly fundamental to develop robotics software. Additionally, the authors discuss the difficulties of guaranteeing adequate automated analysis of complex properties. Finally, they emphasize the need of future reasearch within this formal verification context, due to its role in software development.

For instance, a natural way of performing analysis over a system is by observing the communication architecture. ROS provides a framework for GUI development plugin called *rqt_graph*[6], in which ROS developers usually rely on. It is used to depict the network architecture through a graph with run-time statistics. However, this approach can only be used to aid in the manual detection of potential architectural flaws, so it lacks on providing trustworthy analysis over more complex networks.

The noteworthy High-Assurance ROS (*HAROS*) framework, initially proposed by Santos et al. in *A framework for quality assessment of ROS repositories* [58], holds great value thanks to its contribution in improving ROS's software quality. HAROS makes use of several analysis techniques to perform quality evaluation of ROS software, followed by ways of feed-backing inconsistencies using predefined code metrics. As this framework seeks to be flexible when it comes to adding functionality, further static analysis can be incorporated as plugins [59, 60].

Following these continuing changes on HAROS, Santos, Cunha, and Macedo describe the framework in the tool paper *The High-Assurance ROS Framework* [61], due to the implementation of several plugins that enable other sort of analysis, rather than just static code analysis. The authors explain why covering a wide range of analysis techniques is essential, stating that adopting one technique in isolation might not be ideal nor sufficient.

To put it briefly, in HAROS (Figure 22), the user provides a YAML-based project file, which specifies the configuration launch files that represent concrete applications to be analyzed. Additionally, behavioural properties can be specified to be evaluated during the analysis phase.

HAROS then proceeds to locate ROS-based packages, where source code is extracted and subsequently parsed, yielding a complete graph-based model. These models not only provide developers with visual feedback of the application architecture, but also enable automatic analysis. The analyzing process is carried out by any installed analysis plugins. The visualizer exports JSON data files in order to be capable of providing reported feedback to the user.

Lastly, Santos, Cunha, and Macedo proceed to address some future considerations of HAROS, where the latter is expected to extend both its domain to cover both ROS and ROS2 packages, and its supported plugins.

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

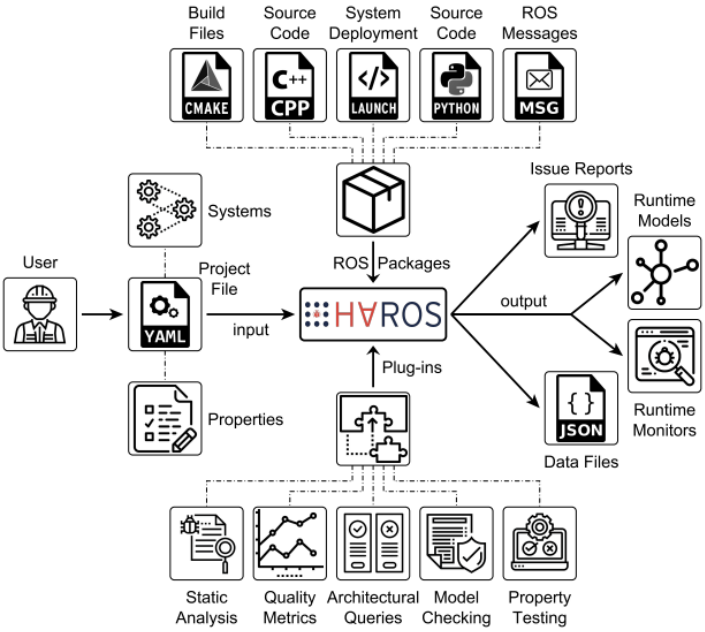

Figure 22: Workflow Architecture of the HAROS framework. Extracted from [61].

The literature concerning property verification has already been explored through state-of-art model checkers, where safety properties regarding the robotics domain were studied and verified. Within the ROS context, certain techniques were proposed that primarily focused on modeling the ROS node-communication with the goal of supporting model-checking of desired properties.

Within this context, Halder, Proença, Macedo, and Santos provide a model-based approach to reason about the communication between nodes in ROS-based systems, emphasizing the specification of real time properties. Here, the authors model ROS applications by abstracting them as timed automata. Property verification is then ensured by model checking through the *UPPAAL*[7] model checker.

Notably, this work provides a brief introduction to the concepts of timed automata and their implementation into UPPAAL, with respect to the latter's temporal logic as a query language to describe desired properties. After contextualizing the code behind a ROS-based application, the authors proceed to perform model checking using the source code of the popular physical robot *Kobuki*[8]. This allowed the authors to reason about context specific properties of the latter, aside from the safety properties.

Halder, Proença, Macedo, and Santos conclude their work by emphasizing the importance of model-checking in finding and checking these desirable properties, stating if only physical verification was considered, the latter's properties would be extremely difficult, time-consuming or infeasible to find.

A remarkable work that duly fits under this dissertation domain of exploration is presented in [11]. Here, Carvalho et al. present a proposal to automatically verify system-wide properties of ROS applications at static time,

7 https://uppaal.org/
8 http://kobuki.yujinrobot.com/

emphasizing their safeness behaviour. Here, the authors abstract each application behaviour through concrete models, to then perform model-checking over system-wide specifications, using *Electrum*[9] as model checker.

In order to integrate analysis of ROS-based applications, the authors make use of the already mentioned HAROS plugin-based framework, that provides quality assessment of ROS software. As early explained, HAROS allows the automatic extraction of the system architectural model from the source code and launch file configurations. Then, using a domain-specific property-based language, it allows the specification of the desired properties and assumptions about the nodes behaviour.

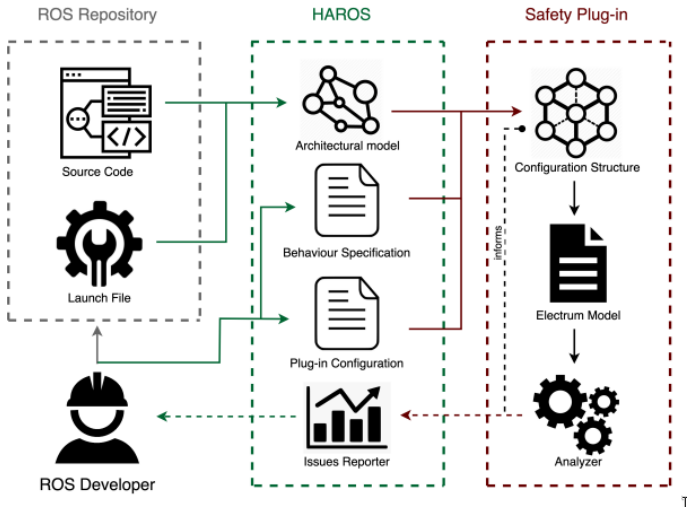

Figure 23: Architecture of the HAROS safety plugin for ROS. Extracted from [11].

The backend of the depicted architecture (Figure 23) relies on Electrum, the former version of the current Alloy Analyzer, to model and reason about the application behaviour through properties expressed in FOTL. Afterwards, the authors pretend to extend *HAROS*' functionality by offering a plugin that automatically converts HAROS artifacts into Electrum. Additionally, the framework reports model checking counter-examples in a more user-friendly manner via its quality reporting interface.

Furthermore, the authors apply their approach to a real ROS-based robot. The presented results show that the specification language is sufficiently expressive to address relevant system's properties. Then, they address the scalability by performing a time evaluation where different values are tested. As future considerations, the authors propose techniques to help discard false-positive counterexamples, that can possible emerge due to the improper scope's configuration, since no particular scheduling is imposed on the message-passing process.

As ROS2 domain regards the use of DDS communication protocol, a few works on DDS modelling analysis deserve to be mentioned, as they might give important background for property verification over communications protocols.

In [1], it is proposed a technique to model the DCPS architectural design that DDS makes use of, alongside with new approaches to the current DDS behaviour. Supported by several modelling techniques for publish-subscribe

9 http://haslab.github.io/Electrum/

systems, in [43], DDS in ROS2 is formalized as a timed automata, consequently followed by model verification over property-checking.

In [71], White, Caiazza, Jiang, Ou, Yang, Cortesi, and Christensen continue to address security [69, 70] through an alternative approach. Here, the authors makes use of formal verification and model checking to reason about reachability of information flow, related to an attack model that studies the network through targeted attacks. This study reviews the DDS security protocol and its model, and concludes that it lacks support for protecting against some security threats.

Within this scenario, White et al. intend to explore the data flow semantics to address the DDS networks vulnerability. After conceiving the threat model, some attack assumptions are explicitly defined over the attack model. Unlike previous network reconnaissance methods and considering the assumptions discussed in the attack model, this approach allows an attacker to build the topology through a graph based database by passively sniffing packets inside the network. This database is then made available to the SAT solver in order for it to compute queries. *Imandra*[10] model checking tool is then used as SAT solver by replicating the DDS security protocol functionalities (especially the access control) as functional models, to accurately represent the default plugin logic. Here, the attack model assumptions are specified as SAT-based queries, allowing the replicated models to reason about them. Moreover, using formal verification and model checking, vulnerability excavation can be efficiently performed over the inquired graph, instead of engaging with the targeted system. The usage of formal verification techniques can reason about a large state space without exhaustive search.

Another notable work regarding former analysis using Alloy specification language is presented in [45], where a safety-critical scenario is proposed under the domain of surgical robots. The formal techniques used allows overview over a surgical robot arm, taking into consideration possible violations of important safety properties. Although this study presents favorable outcomes, its focus lies on a particular area of study. As a result, it lacks on providing solutions to a vast majority of situations.

10 https://www.imandra.ai/

4

RESEARCH PLAN

So far, the work has been mostly based on research, namely exploring the concepts behind ROS2, the deployment of security, and the Alloy specification language. From now on, the dissertation will focus on developing a technique that uses Alloy to support the formal verification of ROS2 environments in regard to security properties.

In the next first month security-related discussions will take place as major priority, due to its critical role on what this dissertation aims to tackle. The main goal is to find a concrete ROS example that fails to address a specific security property, where the latter would be previously evaluated in a speculative manner.

Later, during the *Core Techniques Definition* phase, ROS2 applications architectures will be formalized in Alloy, also taking into account the above's considerations about security and SROS2 configuration. A technique to specify and verify information-flow security properties on top of the Alloy formalization will be proposed.

The *Evaluation* will account relevant case studies in the *Autoware* platform, in order to evaluate the effectiveness of the proposed formalization and verification technique. During the *Implementation* task, it will be developed a prototype tool that can automatically infer an Alloy formalization of a ROS2 architecture and SROS2 security configuration. The writing of this dissertation will start in April and carry out until July.

TASKS	February	March	April	May	June	July
SROS Security Discussion						
Core Techniques Definition						
Evaluation						
Implementation						
Writing						

Figure 24: Dissertation's schedule.

The following work will take place on a 6 month period between the 1^{st} of February 2022 until the 31^{th} of July 2022. The whole timetable is displayed above in the Figure 24, with the times evenly split among the several periods.

BIBLIOGRAPHY

[1] Alaa Alaerjan, Dae-Kyoo Kim, and Dhrgam Al Kafaf. Modeling functional behaviors of dds. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–7. IEEE, 2017.

[2] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[3] Thomas A. Alspaugh. The alloy quick reference. https://www.ics.uci.edu/~alspaugh/cls/shr/alloy.html, 2010.

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

[5] Kai Beckman and Jonas Reininger. Adaptation of the dds security standard for resource-constrained sensor networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4, 2018. doi: 10.1109/SIES.2018.8442103.

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

[7] Benjamin Breiling, Bernhard Dieber, and Peter Schartner. Secure communication for the robot operating system. In *2017 annual IEEE international systems conference (SysCon)*, pages 1–6. IEEE, 2017.

[8] Lionel Briand and Yvan Labiche. A uml-based approach to system testing. In *International Conference on the Unified Modeling Language*, pages 194–208. Springer, 2001.

[9] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. Formal software design with alloy 6. https://haslab.github.io/formal-software-design/index.html#formal-software-design-with-alloy-6, 2021.

[10] Gianluca Caiazza, Ruffin White, and Agostino Cortesi. Enhancing security in ros. In *Advanced Computing and Systems for Security*, pages 3–15. Springer, 2019.

[11] 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)*, pages 7249–7254, 2020. doi: 10.1109/IROS45743.2020.9341085.

[12] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[13] Edmund M Clarke. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[14] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[15] Lori A Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, pages 488–491, 1976.

[16] Agostino Cortesi, Pietro Ferrara, and Nabendu Chaki. Static analysis techniques for robotics software verification. In *IEEE ISR 2013*, pages 1–6. IEEE, 2013.

[17] Steve Cousins. Welcome to ros topics. *IEEE Robotics & Automation Magazine*, 17(1):13–14, 2010.

[18] Steve Cousins. Exponential growth of ros. *IEEE Robotics & Automation Magazine*, 1(18):19–20, 2011.

[19] John J Craig. Introduction to robotics. 2005.

[20] Fast DDS. Dds architecture. https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html, 2019.

[21] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8514–8521, 2019. doi: 10.1109/ICRA.2019.8794451.

[22] Ruisheng Diao, Vijay Vittal, and Naim Logic. Design of a real-time security assessment tool for situational awareness enhancement in modern power systems. *IEEE Transactions on Power systems*, 25(2):957–965, 2009.

[23] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482, 2016. doi: 10.1109/IROS.2016.7759659.

[24] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass, and Peter Schartner. Security for the robot operating system. *Robotics and Autonomous Systems*, 98:192–203, 2017.

[25] Bernhard Dieber, Ruffin White, Sebastian Taurer, Benjamin Breiling, Gianluca Caiazza, Henrik Christensen, and Agostino Cortesi. Penetration testing ros. In *Robot Operating System (ROS)*, pages 183–225. Springer, 2020.

[26] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3): 1729881418770011, 2018.

[27] euRobotics. Why is robotics important. https://www.onlynaturalenergy.com/why-is-robotics-important/, 2021.

[28] Kyle Fazzari. Ros 2 dds-security integration. https://design.ros2.org/articles/ros2_dds_security.html, 2020.

[29] DDS Foundation. About dds. https://www.dds-foundation.org/what-is-dds-3/, 2021.

[30] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

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

[32] Software Design Group. Alloy 6. https://alloytools.org/alloy6.html, 2021.

[33] 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.

[34] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

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

[36] Daniel Jackson. Alloy: a language and tool for exploring software designs. 62:66–76, 2019. doi: 10.1145/3338843.

[37] Merike Kaeo. *Designing network security*. Cisco Press, 2004.

[38] Jongkil Kim, Jonathon M Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. Security and performance considerations in ros 2: A balancing act. *arXiv preprint arXiv:1809.09566*, 2018.

[39] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(268-272):30, 1994.

[40] Savas Konur. A survey on temporal logics. *arXiv preprint arXiv:1005.3199*, 2010.

[41] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[42] Man Lin, Li Xu, Laurence T Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. *IEEE Transactions on Industrial Informatics*, 5(1):22–37, 2009.

[43] Yanan Liu, Yong Guan, Xiaojuan Li, Rui Wang, and Jie Zhang. Formal analysis and verification of dds in ros2. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–5. IEEE, 2018.

[44] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. page 373–383, 2016. doi: 10.1145/2950290.2950318.

[45] 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, 2018.

[46] Waleed Mansoor. Understanding ros nodelets. https://medium.com/@waleedmansoor/understanding-ros-nodelets-c43a11c8169e, 2018.

[47] Gerald A Marin. Network security basics. *IEEE security & privacy*, 3(6):68–72, 2005.

[48] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[49] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascarenas. A preliminary cyber-physical security assessment of the robot operating system (ros). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013.

[50] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pages 736–742. Ieee, 2008.

[51] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking. In *International Static Analysis Symposium*, pages 330–354. Springer, 1999.

[52] Object Management Group (OMG). Data distribution service (dds). https://www.omg.org/omg-dds-portal/.

[53] Object Management Group (OMG). Dds security. https://www.omg.org/spec/DDS-SECURITY/1.1/, 2018.

[54] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to dds and data-centric communications. *Real-Time Innovations*, 2005.

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

[56] Open Robotics. The robot operating system. https://www.ros.org.

[57] Open Robotics. Ros 2 documentation. http://docs.ros.org/, 2021.

[58] 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.

[59] 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.

[60] 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.

[61] André Santos, Alcino Cunha, and Nuno Macedo. The high-assurance ros framework. *arXiv preprint arXiv:2103.01603*, 2021.

[62] OMG Specification. The real-time publish-subscribe protocol (rtps) dds interoperability wire protocol specification. *Object Manag Group Pct07-08-04*, 2007.

[63] George Stavrinos. Client libraries. http://wiki.ros.org/Client%20Libraries, 2020.

[64] Shu Takemoto, Kanata Nishida, Yusuke Nozaki, Masaya Yoshikawa, Shinya Honda, and Ryo Kurachi. Performance evaluation of caesar authenticated encryption on sros2. In *Proceedings of the 2019 2nd Artificial Intelligence and Cloud Computing Conference*, pages 168–172, 2019.

[65] Amirhossein Vakili and Nancy A Day. Temporal logic model checking in alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 150–163. Springer, 2012.

[66] Thomas Wahl. Fairness and liveness.

[67] Ruffin White and Mikael Arguedas. Ros 2 security enclaves. https://design.ros2.org/articles/ros2_security_enclaves.html, 2020.

[68] Ruffin White and Kyle Fazzari. Ros 2 access control policies. https://design.ros2.org/articles/ros2_access_control_policies.html, 2021.

[69] Ruffin White, Dr Christensen, I Henrik, Dr Quigley, et al. Sros: Securing ros over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060*, 2016.

[70] Ruffin White, Henrik I Christensen, Gianluca Caiazza, and Agostino Cortesi. Procedurally provisioned access control for robotic systems. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[71] Ruffin White, Gianluca Caiazza, Chenxu Jiang, Xinyue Ou, Zhiyue Yang, Agostino Cortesi, and Henrik Christensen. Network reconnaissance and vulnerability excavation of secure dds systems. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 57–66. IEEE, 2019.

[72] William Woodall. Ros 2 on dds. https://design.ros2.org/articles/ros_on_dds.html, 2019.

[73] Meng Wu, Haibo Zeng, Chao Wang, and Huafeng Yu. Safety guard: Runtime enforcement for safety-critical cyber-physical systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[74] Seongjin Yun, Jun-Hong Park, and Won-Tae Kim. Data-centric middleware based digital twin platform for dependable cyber-physical systems. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 922–926. IEEE, 2017.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.