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

Following the current growth of cyber-physical system, as well as their usage into various technology domains, the development of software applications is demanding due to the complexity behind the integration of needed services, beyond those provided by the operating system. Hereupon, 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 DDS-Security specification. Using DDS-Security, it is possible to configure ROS2 to run with security guarantees using the SROS2 toolset [27].

This dissertation proposes a technique, based on the software verification perspective, to automatically verify system-wide properties related to the security configuration of ROS2-based applications. The intended purpose is to model the ROS architecture, as well as the network communication behaviour, in Alloy [31], a formal specification language and analysis tool supported by a model-finder over which, system-wide properties are subsequently model-checked.

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

RESUMO

A constante implementação da ideia de automização de processos tem motivado a reestruturação nos mais diversos setores industriais, com o objetivo de aumentar a eficiência e precisão nos processos integrados, consequentemente, reduzindo os custos associados. Além disso, esta ideia impulsiona a integração robótica nos mais amplos domínios tecnológicos, especialmente em domínios 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 de sistemas ciber-físicos, nomeadamente sistemas robóticos, bem como sua utilização em vários domínios tecnológicos, o desenvolvimento de aplicações é exigente 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, a constante priorização na flexibilidade resulta num aumento de vulnerabilidades de segurança, pondo em causa a integridade das aplicações. Além da falta de segurança apresentada, 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, onde diversas implementações DDS usufruem de tais 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 SROS2 *toolset* [27], é possível configurar o ROS2 de forma a adotar estas medidas de segurança.

Esta tese propõe uma técnica para a verificação automática de *system-wide properties* em aplicações ROS. Esta técnica apresentada baseia-se na formalização estrutural de arquiteturas ROS em Alloy [31], com o obejtivo de modelar o comportamento associado à comunicação dentro do sistema, tendo em consideração configurações associadas às 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

The concept behind automation development is 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, as it is significantly reliant on a variety of technologies. Due to the continuous growth of technology in these different domains, robots can be used in a wide range of applications [49], since their usage brings increased productivity, safety and more manufacturing production work back to developed countries [26, 17].

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

Since robots became to be integrated into distributed systems through separated components, connecting different hardware and software modules raises interoperability and communication issues. To solve this issue, 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 [49, 47]. The requirement for a middleware layer that meets different robot's specification is a novel approach to enable the creation of robot applications over robotic systems, while supporting features such as robustness and modularity.

The Robot Operating System was created by a collaborative open-source community to contribute in the advancement of robots, with the aim of helping build robot applications easily [25]. It enables locomotion, manipulation, navigation, and recognition tasks over complemented software libraries and tools. Concerning the wide range of robotics hardware and software, ROS was designed to be flexible, enabling interpolation with potential added 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 safety-critical and real-time systems [37].

Besides having no middleware support for distributed real-time systems, security was not prioritized by ROS, which started to be demanding for deployed systems. An increasing number of real-time applications, for instance robotic systems, requires security insurance for protecting real-time sensitive data against unauthorized access [41].

This lead to the creation of Robot Operating System 2, developed using the Data Distribution Service (DDS) [52] specification protocol as its middleware, leveraging for its messaging architecture. 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, providing security guarantees to the deployed robotic systems [6].

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, Alloy [35, 43] 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, where the system is abstracted as a conventional model, that is automatically checked over performing property verification on finite-state machines [7].

The proposal of this dissertation is to develop a novel technique to automatically verify system-wide safety properties using Alloy framework, confining a ROS2 system into an abstract model, in order to 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 rests in introducing concepts around the Robot Operating System, contextualizing the evolution behind its framework towards achieving security, where 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 is evaluated, 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.

Security configuration related to SROS2 toolset will be provided in this chapter, supported by an example that accounts multiple security features, those being authentication, encryption and, most importantly, access control, applying restriction constraints to the network and its participants, that by default are not controlled.

The second goal is to extend a previously proposed [10] formalization of ROS applications in Alloy [31, 43] to also take into consideration the security configuration defined with SROS2. Using this extension, we intend to explore the viability of verifying simple information-flow security properties. 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 current dissertation structure is divided into three different chapters. Chapter (2) introduces all the concepts related to Robot Operating System, and its evolution as robotic development framework towards achieving system security. Chapter (3) introduces the Alloy framework, as its specification language supported by a concrete example case. Chapter (4) presents previous developed work that covers concepts that are also addressed within this dissertation, as well as, expected considerations about the future work.

2

SOFTWARE DEVELOPMENT IN ROS2

Robotic systems have emerged into several scenarios, where its usage ranges between basic processes automation, up to full performance over critical tasks, consequently causing the complexity increase in these domains.

Due to the wide variety of robotic hardware presented in multiple domains, concerning about software development is rather difficult [16]. The reuse of code is non-trivial, and therefore, large-scale development is rendered 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 insurance must be addressed as a developing priority [25, 37].

The upgraded 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 middleware implementations are built over this standard, which provides numerous DDS-based specifications as well as valuable Quality of Service (QoS) transport parameters.

The DDS-Security specification [53] 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 over the major concepts on which this thesis rests. First, it is presented 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. Regarding this goal, Data Distribution Service (DDS) and its integration on Robot Operating System 2 (ROS2) must be contextualized beforehand.

2.1 ARCHITECTURE CONSIDERATIONS

The Robot Operating System was created by a collaborative open-source community, that has undergone rapid development [16] to contribute in the advancement of cyber physical systems. It was purposefully designed to be a development enhancer for the realm of robotic applications [25, 55].

Fundamentally, ROS is a middleware, as it provides 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 [11], allowing ROS applications to be built over several software modules, as independent computing processes called *nodes*. These compose together to fulfill the deployment characteristics of the corresponding robot [47].

2.1.1 *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 architecture 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. Consequently, every information exchange within the network had to go through this master.

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.

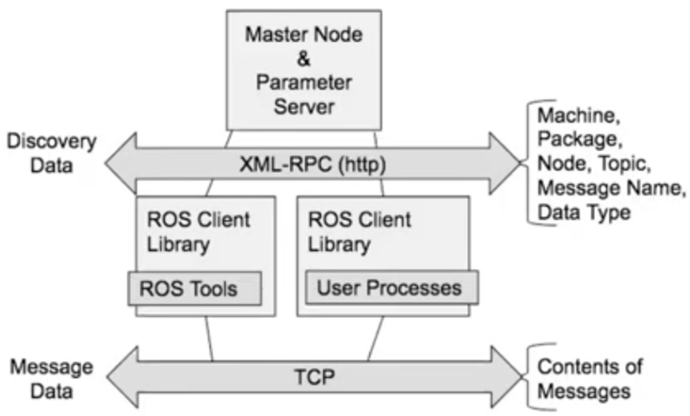


Figure 1: Robot Operating System architecture.

Many research communities tried to fix these real-time issues by proposing potential solutions, while supporting the same architecture design. Unfortunately, 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 [47].

The *Robot Operating System 2* comes as a complete refactoring of ROS, with the aim of increase the framework's real-time capabilities, by allowing the development of time-critical control over ROS, as it moves away from the former architectural design towards the implementation of an external middleware that can support the production needs of the outgrowing robotic systems [37, 11].

2.1.2 *Data Distribution Service*

The *Data Distributed System* (DDS) [52] is an *Object Management Group* (OMG) middleware standard. The standard was developed to address the demand for enhanced interoperability across different vendors' middleware frameworks, directly addressing data communication between nodes that belong to a *publish-subscribe* communication architecture, 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 [28].

DDS 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 quest 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 [47, 2, 19].

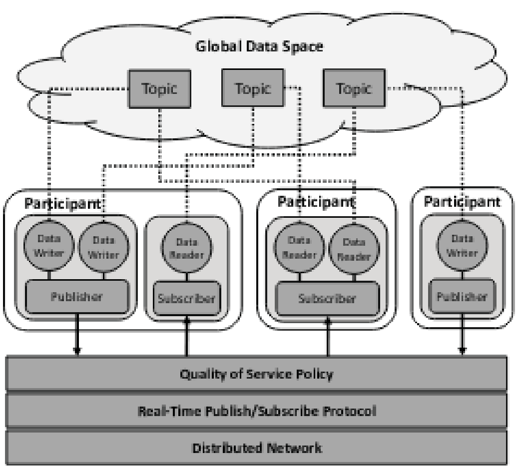


Figure 2: DDS architecture: DCPS model with RTPS. Extracted from [47].

To properly address the data transportation through physical network, DDS offers a wire specification protocol called *Real-Time Publish-Subscribe Wire Protocol* (RTPS) [61], 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 [19, 2].

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 [2, 25, 47].

Briefly, DDS leverages the premise of a transport-independent virtualized *Data Bus* 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.

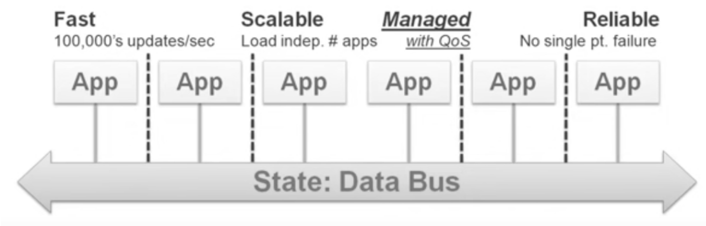


Figure 3: Data Distributed System architecture in a nutshell.

2.1.3 *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 approach is built upon the DDS framework [47], leveraging DDS for its messaging architecture, where communication and transport configuration are handled.

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

The middleware's on-top layer regards the ROS client library (*rcl*), already implemented in the former ROS architecture. This layer accounts the availability of ROS concepts to the Application layer, as it provides APIs to ease the software implementation by ROS developers [57]. As ROS aims to support different programming languages over the same computing context, each language-specficic API must have its corresponding client library (*rclcpp* regarding *C++* and *rclpy* regarding *Python*). The *rcl* accounts these client libraries by abstracting their specification, consequently reducing code duplication [62, 11].

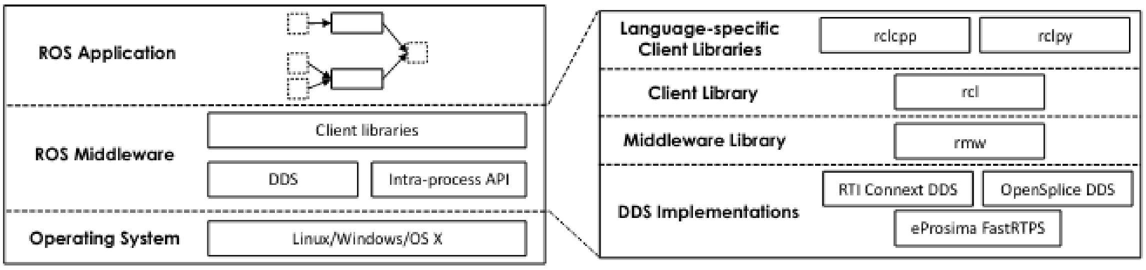


Figure 4: ROS2 framework architecture.

Towards supplying a wide range of configurations back to application layer, ROS2 aims to support multiple DDS implementations, in which these implementations API specification might differ from each other (currently, *FastRTPS* by *eProsima*, *Connext* by *RTI*, and *Vortex OpenSplice* by *Adlink*). It should be noted that the DDS implementations are 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 the specifications complexity of these implementations APIs, an DDS-agnostic interface is being introduced, the *rmw* (ROS MiddleWare) interface [11], allowing portability among DDS vendors, which consequently enables ROS developers to interpolate DDS implementations, based on their applications needs during runtime. The information flow through the middleware layer is done over structure mapping between ROS and DDS data models, addressed by the *rmw*, regarding the DDS implementation that is being considered at runtime.

2.1.4 *Computation Graph*

From a logical perspective [11], 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 publish-subscribe model approach as communication type, through *message-passing* patterns, confers additional concept complexity to the application architecture, where the latter can be naturally represented as a *computation graph* [15].

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 over package orchestrating, where each logically represents a useful software module. Packages might be compromised 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, as they are built over their corresponding packages [15, 55].

Thus, the network is comprised by many nodes, running simultaneously and exchanging data between them, where each node addresses its corresponding network module 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 when developing a complex robotic system.

The notable usage of callback functions provide 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 variables [57, 55].

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

The concrete mechanisms *Topics* are perhaps the most common method, naturally perceived as middle-communication buses, over which messages are passed through. As semantic approach, communication through topics is handled by the publishing-subscribing pattern. 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 [11].

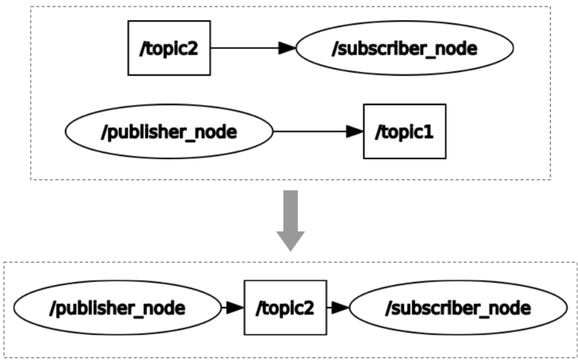


Figure 5: Communication behaviour over *topics*.

A specific *topic* is created upon specifying its entity name over either a publisher or a subscriber callback instance. Whenever a node creates a publisher, intentionally instantiated to publish a message 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 receipt, as it can also be utilized for publishing purposes [11].

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, furnish 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 receiving of this goal. The server might provide feedback to the client before providing a response to the client.

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. Therefore, 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.

The Figure 6 depicts the network architecture corresponding to an ROS application well-known example called the *TurtleSim*. This application is mainly composed by *two nodes*, that perform together towards moving a turtle. Additional nodes were implemented, in order to add complexity to the current network, as to later support security as a proper example. 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. 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* [1], *rqt_graph*, allows the developer to perform analysis over a graphical visualization of the network computation graph.

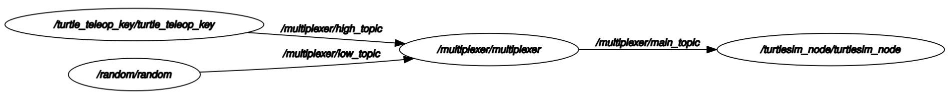


Figure 6: *TurtleSim*'s network graph presented by *rqt_graph*.

After the proper configuration of each node regarding the *TurtleSim* example, the network can be easily managed and automatically launched through a launch file. The Figure 7 addresses the launch specification related to the latter application.

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

```
<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 7: *TurtleSim* launch file.

Additional node configuration, such as name remapping and parameter adjustments, can be specified under the *args* tag, which offers great functionality to the launching process.

Distinctive namespaces allow the system to start the nodes, without any name nor topic name conflicts. However, this technique has some flaws attached, since 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 nodes *parameters*, that allows individual configuration of the network nodes. In the former version of ROS, the node parameters were controlled by a global *parameter server*, managed by its 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 parameter function 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 approaches 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 [45]. Supported 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].

2.2 SECURITY

The deployment of real-time systems implies critical concerning about safety and security [47], 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 [48, 22].

2.2.1 *Former ROS Security Concerns*

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 [36]. The appliance of security countermeasures techniques upon configuring the system's network confers a critical step when aiming towards achieving security.

Within this vast topic, several avenues of endeavor come to mind, each deserving of a substantial study. Network security entails pre-exploration of the system's network through practical networking security techniques, such as intrusion detection and traffic analysis [46].

Numerous researchers [20, 24] 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 to ensure protection over the Master node. Exposing this node poses a critical threat over the whole network [20].

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 settled in plain text, making data content vulnerable to unauthorized usage [37].

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 [21].

2.2.2 *DDS-Security Specification*

The *Object Management Group* (OMG) [52] accounts security integration by supplying an in-depth specification, consequently adding features to the already 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 [6, 27].

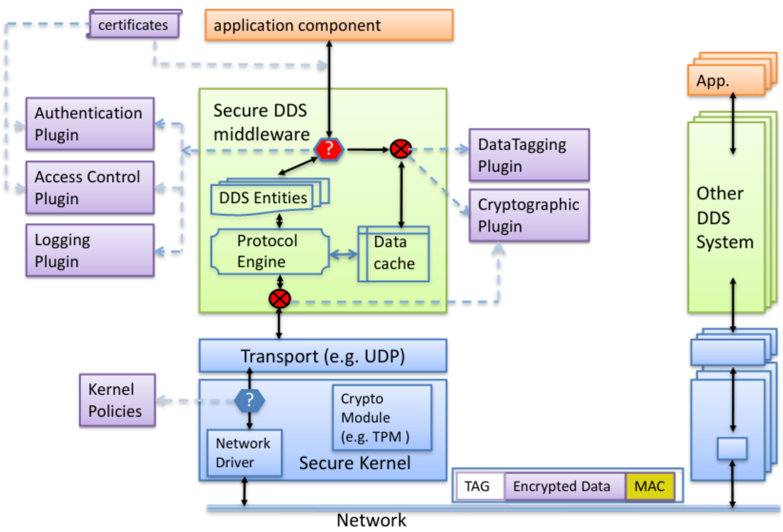


Figure 8: DDS-Security Architecture. Extracted from [53].

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 each DDS domain participant.

The Authentication plugin confers the most valued plugin to the entire SPI architecture, as it provides means to validate the identity of the application, later regarded as a domain participant [53, 27].

Each participant must be authenticated prior to entering the data space [68]. Therefore, participants are presented to the secured environment regarding the Public Key Infrastructure (PKI). This latter is in charge of issuing a public certificate, accountable and signed by a trusted certificate authority (CA) [68, 69, 27].

The communication establishment over different participants must be preceded by a mutual handshake, where certificates are exchanged to guarantee their authenticity [68, 37]. Additionally, the DDS permissions of a domain peer are also concerned within 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 respective operations are imposed by policy restrictions

[53, 68]. 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 [27].

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 [68].

Following the authentication procedure, domain authorization is also concerned using the proven Public Key Infrastructure (PKI) [27], by embedding policy definitions through certificate extensions [69].

Furthermore, the Access Control plugin employs *2* configuration documents that are allocated to each participant [68]. 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) [27]. 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 commence [69].

Communication and Encryption

The DDS-Security specification ensures encryption and authentication using *OpenSSL*, while accounting security functions based on encryption standards [63].

Accordingly, it implements a handshake-based standard, concerning the *OpenSSL* protocols, *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS), which are used respectively used to ensure encryption over the network communication [69, 37]. The handshake is used to achieve mutual authentication within participants over the DDS domain [68].

Following the public key assignment, the *Diffie-Hellman* key exchange protocol properly accounts the mentioned handshake, allowing both participants to exchange data over a shared secret key, while accounting their own public certificate information [37]. 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 concerning data encryption over the implicit channel [37, 63].

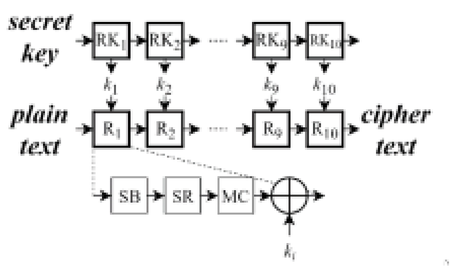


Figure 9: *Advanced Encryption Standard* algorithm. Extracted from [63].

The *key cryptosystem* algorithm [63] presented in the *Advanced Encryption Standard (AES)* considers the usage of functions towards achieving data encryption over the established communication.

Here, the algorithm combines the shared key established with the message passed over the secure channel. Moreover, it is desirable to implement the *Galois MAC (AES-GMAC)* encryption algorithm, that is based on block cipher operations, consequently adding encryption functionality to the AES algorithm, through a *Message Authentication Code (MAC)* encryption function [63, 37].

2.2.3 *Security Integration in ROS2*

As result of the *Data Distribution Service* (DDS) implementation as a flexible middleware interface in the ROS2 architecture, issues regarding security is 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 [37, 20].

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 [27], ROS2 provides a toolset, the *Secure Robot Operating System 2* (SROS2) toolset [1], extending ROS2's functionality to make use of the DDS-Security functionality.

The control over these tools are done by *rcl*, providing security over the Application layer, while DDS is capable of providing security over the communication architecture [37]. SROS2 configuration is done over applying a set of security files to each ROS2 participant, with regard to how DDS handles certificate assignment to their participants [69].

The variety of capabilities in SROS2 toolset attempts to aid with security configuration across environments [27]. 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 process [27, 68].

Security Enclaves

The authentication process within the ROS network relies on the notion of a network enclave. 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.

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* [68].

Accordingly, ROS2 ensures authentication over nodes by conceiving the idea of security enclaves, where security artifacts are stored to properly achieve security over the network data space [67]. Recall that, these artifacts are implicitly concerned by DDS-Security specification, where the Certificate Authority and an established Public Key Infrastructure (PKI) comes in hand [68, 69].

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, as memory space becomes rather difficult to handle [67, 71].

Concerning the enclave authentication procedure, its security artifacts must be addressable by a DDS participant, where the latter matches to a node sharing context [67].

Access Control within Enclaves

Following the *ConArmor* policy language [70], the SROS2 toolset offers a *XML schema*, 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. Typically, each *enclave* declaration is linked to a corresponding ROS node, naturally perceived as a DDS participant.

Objects are classified over a subsystem type, structurally characterized by permissions tags. Then *object privileges* are controlled over access values, either *allow* or *deny*, attributed to their corresponding permissions tags [71]. The policy design approach works under the *Mandatory Access Control* (MAC), that denies any privilege by default. The only way of allowing access to any object, is by explicitly specifying the subject's privilege access [71, 70].

Depicted in the Figure 10, it is presented a policy file where access control privileges are distributed across enclaves, and their inherited profiles. Recall the *TurtleSim* example, the following *XML* policy file addresses the access to topics for each respective enclave.

```xml
<?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 10: SROS2 policy file regarding the access control policies over the *TurtleSim* example.

3

ALLOY SPECIFICATION FRAMEWORK

As aforementioned, this dissertation aims to tackle the security vulnerabilities resulted from the miss-configuration over ROS files. In this chapter, it is intended to explore the Alloy framework that is relevant to overcome the above-mentioned challenge.

The increasing usage of robotics onto safety-critical systems results in demanding considerations over ensuring the proper correctness of both software and hardware, as failures mainly regarding the security domain might lead to fatal consequences. 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 [13]. Software frameworks designed for this purpose must provide methods to perform structural design over systems with rich structures, abstracting their behaviour as a conventional model. Additionally, these frameworks must support features to enable automate analysis, in which property evaluation over these designed models is used as technique.

The *Alloy Framework* [35], fits within this context, as it furnishes a declarative relation-based language, used for software modelling, complemented with extended tools supporting analysis over these models [31]. 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 over the specified model, Alloy performs model-checking techniques over these logic languages, where the model M is exhaustively checked over property verification [43].

The framework *analyzer* takes the specified model's restrictions into account, performing *Bounded* and *Unbounded* Model Checking to find instances that satisfy those implied restrictions. It can be also be useful for checking model properties, where the analyzer will try to return a counterexample instance. Instances are displayed by the framework Visualizer, alongside with the modelling process steps, regarding a trace representation. Instances appearance can be customized, using the *theme*'s extension [31].

This chapter will go through these principles in further depth, to give the reader a proper review on how Alloy is structured, as its importance as a model checker to the computation domain, supported by a previously configured example where ROS communication architecture is structurally modelled. Since system analysis rely on the reasonable implementation of Model-Checking techniques, the following section within this subject intends to cover a clear contextualization on this matter.

3.1 MODEL CHECKING

Performing software testing has been regarded as the established assessment procedure in which functional and non-functional specifications are evaluated. The conventional approach on software verification is based on testing the system with different inputs, to achieve quality assurance over several intended specifications [7, 8]. As this technique demands exhaustively evaluation over pre-selected test data, it is commonly explored over automated tools, since manual testing is time-consuming and prone to errors [14, 29].

Model Checking presents itself as a novel technique with the purpose of verifying temporal properties over the 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 [29, 7]. This technique is becoming highly used due to its importance as an early phase approach upon developing systems [43], as it confers the most valued functionality over model-checking frameworks, in which concrete models, regarding the software architecture, is exhaustively checked over behavioural properties.

It provides highly automatic verification procedures, where other techniques, such as theorem provers fails to address, due to its deductive reasoning nature. The representation of not satisfied specifications over counterexamples, confers great functionality to this technique, often required for debugging matters. However, the system's inevitable state expansion consequently causes the complexity increase on verification. This is referred to as the *state explosion problem*, in which model-checking is unable to handle the size of the state space [13, 12]. Yet, this is can be mitigated using bounded techniques.

Model Checking techniques accounts 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 must be abstractly represented as a transition system, perceived as concrete models, to perform property checking over the latter, while considering the formula defined in a temporal logic [33, 64].

Transition System

A *Transition System* is defined over a graph-based structure, that confers additional representation over the mathematical graph structure. The latter offers weak ability to provide a concrete system description over a discrete model. Commonly, transition system confers a labelling function that maps each state, naturally perceived as graph node, with so-called *atomic propositions*. These propositions evaluate system variables in each state [50, 64].

Kripke Structures were intentionally conceived to address the model checking field of action, so they naturally fall under the umbrella of this vast domain [50].

Conceptually, a *Kripke Structure* defines 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 p_s enjoyed by it, draw from the domain of p ($p_s \subseteq p$).

Model Checking is a *model-based* technique in which property verification concerns are centered on the concept of *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. As this computing process relies on a state representation, the formula verification also accounts a model state s. $M, s \models \psi$ [33, 50].

Additionally, 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* [33]. 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 relies on the system properties to be analyzed [50], as they confer different model-checking algorithms [33].

3.2 STRUCTURAL DESIGN

The *Alloy framework* presents itself as a formal modelling language, conceived to properly address model-checking techniques over their specification language, where both structural design and temporal behaviour, naturally specified over properties, can easily be defined. Formerly, Alloy was inherently static [43], meaning that it only excel the structural design, where its language was based on first-order logic. The analysis process relied on a bounded model checking technique with no support for temporal behaviour. Notwithstanding, the latest release of Alloy confers the ability to properly deal with expressive temporal properties, as well as trace evaluation over time, while employing the former structural approach.

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 [43]. The following subsection 3.2.1 addresses the Alloy concepts required for understanding how system modelling is covered.

3.2.1 *Structural Modelling*

Alloy aims to address the complexity behind richly structured systems, that require critical control over their intended behaviour, by presenting a novel approach for abstracting these systems as conventional models.

System's structures can be specified over time-evolving states, where its behaviour clearly identifies the states' inbetween 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 the latter is abstracted in terms of relations. In Alloy, unary relations, commonly known as sets, are labelled as *signatures*, that are inhabited by a set of *atoms*, 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, addressing relation between the signature's atoms and a set or other relation. Fields are inhabited by tuples of atoms from the universe, that must meet the same arity.

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

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, where it is commonly used to express singleton sets, over the constraint keyword $one\ sig$. Fields, however, makes great use of multiplicities by restricting behaviour over relations between atoms. In addition to these model constraints, explicitly specified over the course of the modelling process, system assumptions can be defined over axioms, expressed as $facts$, where multiple constraints can be incorporated [30].

Moreover, the latest Alloy version enables evaluation changing throughout the trace evolution, consequently allowing the consideration of both signatures and fields as time mutable declarations, through the usage of the keyword var.

Throughout the sections that follow, it will be presented an illustrative example over which graph theory rests, 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 modelling must be provided beforehand.

Model Structure

With regard to the presented example, as it falls under the study of graphs theory [66], it follows an abstract representation of the graph mathematical structure. In this sense, a graph is made up of nodes which are connected by edges.

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*, that connect together over relations, 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 followed by the corresponding *Node* signature declaration, represents the state of our intended example. 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 upon the *Node* signature declaration. The *adj* concerns 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. The latter represents a mutable relation, identified by the keyword var, meaning that its evaluation may change during the course of the trace's evolution, as opposed to the static ones.

Relation multiplicity constraints are explicitly defined in the field declaration through the use of multiplicity operators, with those being one, $lone$, $some$ and set.

Multiplicity constraints in declarations

Set declarations with multiplicities 76	
e is a expression producing a set (arity 1)	
x: **set** e	x a subset of e
x: **lone** e	x empty or a singleton subset of e
x: **some** e	x a nonempty subset of e
x: **one** e	x a singleton subset of e (i.e. a scalar)
x: e	x a singleton subset of e (equivalent to **one**)

Figure 11: Alloy Multiplicity constraints. Extracted from [4].

Structural constraints can be entailed over explicitly signature declaration. These are often referred as *signature facts*, universally quantified over the signature's set [4]. Suppose a hypothetical design scenario, where the relation *adj* is labelled as mutable. To ensure that the graph architecture consistency, in which, edges are structurally fixed, the following constraint can be specified. Additionally, consider the following *graph* property, where it is desirable to express the following axiom: *The graph contains no self-loops.*

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

Node hypothetical constraints over the signature definition.

As stated, the former poses a model incoherence within the *Node* signature declaration, as *adj* field is formerly labelled as mutable, that is later refuted by specifying $always\ adj' = adj$. This latter introduces the Alloy language's ability to express temporal behaviour through these *two linear temporal logic* operators, $'$ and $always$ respectively.

The $always$ operator expresses a universal quantifier over time, imposing a constraint throughout the trace. The $'$ operator evaluates the *adj* relation in the next state. So, the formula specifies that the evaluation of *adj* in the next state is always the same of the current state. Alloy integrates LTL into the standard Relational logic, therefore, supporting both linear temporal logic unary and binary modal operator.

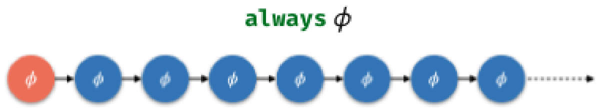


Figure 12: *always* behavioural representation.

Whereas the latter expresses the *no self-loop* graph property. Even though *adj* is introduced as a relation between *Nodes*, it is perceived as a set of *Nodes* 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.

Aside from the structural constraints implied within signature's declaration, multiplicity over fields 11 and signatures also narrows the model's universe. 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.

The above signatures accurately support the *Eurelian* circuit concept. An *Eurelian* path denotes a trail in a finite graph where every edge is visited exactly once. Additionally, an *Eurelian* circuit implies that the path must start and end at the same node, denoted as *Init*. As this latter differs from the remaining *Node* atoms, the *extends* keyword must be used to imply hierarchy disjointness. The *Euler* node is an abstract representation of the current node that is being visited. As this impose a mutable state (*var*) over nodes, hierarchy disjointness is not appropriate. Hence, to properly model that the *Euler* node can be included in an arbitrary atom, signature inclusion (*in*) should be used.

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.

Additional 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 *Eulerian* path denotes a trail within a finite graph, with each graph edge being visited precisely once. Thus, this already implies that the graph must be connected, where each node must be reachable, and undirected, where edges are non-oriented.

```
fact eulerian_considerations {
    adj = ~adj
    no iden & adj
    Node->Node in *(adj + ~adj)
}
```

Graph restrictions through $fact$ declaration.

These expressions, above specified, make use of some valued Alloy operators, consequently identified either as a set-theory operator or as a relational operator. 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	$\Phi \subseteq \Psi$
$\Phi = \Psi$	$\Phi = \Psi$
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 13: Relational Logic Syntax.

Regarding set-theory, set intersection (denoted by &) and set conjunction (denoted by $+$) are introduced. The remaining are relational based operators. The $\sim adj$ denotes the converse relation of *adj*, and the $\rightarrow$ represents the Cartesian product operator. The reflexive transitive closure operator $(*)$ confers the smallest transitive $*(adj\ +\ adj)$ relation containing all the identifiers, reachable in zero or more steps, through the implicit use of the set composition operator $(\cdot)$. The other transitive closure ^ is defined as $\hat{}rel\ =\ *rel\ -\ iden$.

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

Figure 14: Reflexive transitive closure operator.

In order to prevent unwanted model structural scenarios, the intended model can be visualized over the *Alloy Visualizer*, upon executing the run analyzing command. In the Figure 15 is depicted an instance of an acceptable configuration of the *Eulerian* circuit with *5 Node* atoms, since every model constraint was duly specified.

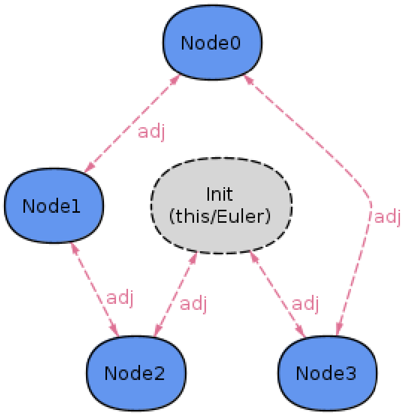
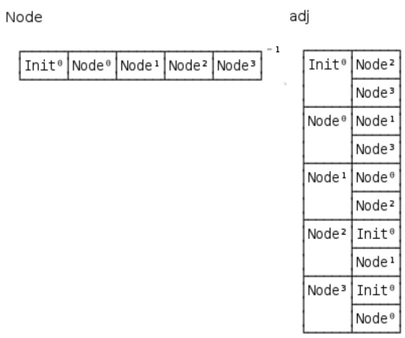


Figure 15: Acceptable *Eulerian* graph model design.

Everything is a Relation

As previously stated, Alloy emphasizes the mathematical relation concept to describe systems as a conventional designed model, through conceiving a set R of relations. Moreover, Alloy confers the ability to express their relations as *variable* [43]. An Alloy relation presents itself as a set of tuples of *atoms* drawn from the same universe context. Subsequently, each relation tuple must meet the same arity of the relation [9].



Node

Init^0	Node^0	Node^1	Node^2	Node^3

adj

Init^0	Node^2
	Node^3
Node^0	Node^1
	Node^3
Node^1	Node^0
	Node^2
Node^2	Init^0
	Node^1
Node^3	Init^0
	Node^0

Figure 16: *Node* and *adj* relations.

The *Alloy Evaluator* [31] 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]. In the Figure 16 is presented both *Node* signature and *adj* relation, regarding the model depicted in the Figure 15.

Additionally, the use of the relational based operators $<:$ and $:>$ denote explicit restriction over relations, with the former restricting its domain, and the latter restricting its range.

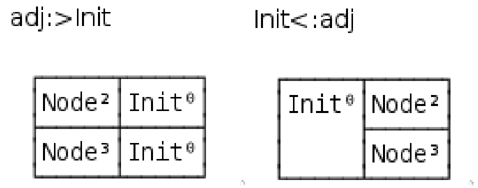


Figure 17: *Init* adjacent nodes.

3.2.2 *Structural Behaviour*

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 explicit specification of axioms, identified using the $fact$ keyword, alongside with the explicit model assumptions covered by each sig declaration, also constrains the system's behaviour [30].

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 behaviour representation over multiple forms through an event idiom [43].

System transitions are declared over events, where each event is conveniently specified in separate *predicates*. The latter, denoted as $pred$, enables Alloy to express boolean formulas that only hold their value when invoked.

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 regards how the system evolves through the next state, providing a valid outcome of its event. The effect must account the mutable variables of the model through the usage of the $'$ operator, to guarantee the non-occurrence of unexpected behaviour. In addition, the temporal operator $after$ can be used to hold the truth of a formula in the next state.

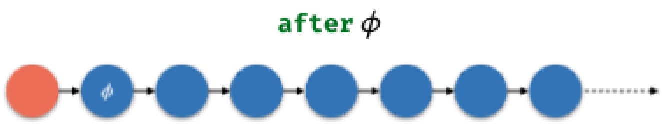


Figure 18: $after$ behavioural representation.

In order to provide a proper contextualization over these concepts, it is desirable to recall the *Eulerian* circuit example. The process of visiting nodes must be specified as it confers the main functionality within the graph theory. It follows the *traverse* predicate to effectively express the latter process.

```
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 states the existence of some node *adjn*, that must be related to the *Euler* node ($Euler \cdot adj$). Another guard is specified to cover the *Eulerian* condition where each edge must be visited exactly once ($adjn\ not\ in\ Euler \cdot visited$). It is followed by *2* effects, where the *visited* relation must account the latest visited edge, and the *Euler* node must be incorporated in the respective adjacent node, within the next state.

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

Eulerian stutter event.

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 evaluation of the trace evolution. The *stutter* predicate captures the "no effect" reasoning approach about the behaviour of the system, where frame conditions formulas are enclosed between braces.

Every conceivable trace inside the system must be a permutation of these event handlers performed on a well-defined initial state. When used consistently, this approach constitutes a model design pattern called as Implicit Operation Idiom [30].

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

Eulerian Initial State.

The *_init* states valid conditions valid in the first state. Thus, to properly specify the initial behaviour of our intended model, every edge must start unvisited ($no\ visited$) and the *Euler* node should start in the *Init* node.

As previously claimed, to reason about state transition, the notion of an execution trace should be duly introduced. The system's permissible behavior will be thoroughly specified by restricting the set of valid operations. As it already been specified every acceptable operation, it is presented the following $fact$ axiom.

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

Trace constraint through the use of an axiom.

The temporal operator $always$ followed by the desired formula is used to impose a state constraint, that must hold one of the *2* events, over the full trace evaluation. This axiom, as explained, must account the initial state (*_init*) conditions.

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, resulting in simpler way of managing specifications [43, 9].

3.3 STRUCTURAL 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 [43]. *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 [43, 39].

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 over a system's design.

3.3.1 *Analysis and Verification*

Alloy specification process does not establish a clear process separation over the model design and model analysis. This implies that analysis over model checking also accounts the model itself as a combination of properties specification.

The specification language holds *two* analyzing commands, run and $check$ respectively, as shown above in *Alloy's syntax*. A *First-Order Linear Temporal Logic* (FOLTL) formula is enclosed between braces, to be checked over. Upon executing, both commands accounts the enclosed formula ψ_f and the model declaration M. Likewise in $fact$ declaration, several logic constraints can be combined into the formula ψ_f. Due to Alloy's implicit abstraction, both commands addresses the relational model specification as truth holder, with the latter being consisted by declarations over $fact$ and sig.

Briefly speaking, run instruct the model-checker to present an example that satisfies the considered formula ψ_f over the model definition. This means that the following formula ($M \wedge \psi_f$) is expected to hold. The consistency of the *facts* and *signatures* is consequently 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 over proving if the following satisfiability formula $M \models \psi_f$ is valid within the defined scopes.

As result of the first-order logic's undecidability problem, the consistency proofness over satisfiability formulas is not possible [64]. To ensure decidability, the Alloy Analyzer performs analysis over *scopes*, assigned to each signature declaration. By default, if no scope is provided, the model-checker will account at most *three* atoms for each signature, upon trying to provide an *instance* that satisfies the former formula, consequently proving the behaviour specification (*facts* considerations) consistency. Also, as scopes imposes a limit on the state space of an Alloy model, if no run instance is returned, the formula can not be perceived as inconsistent, as scopes can not be properly set to satisfy the latter.

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

Figure 19: FOLTL formula ψ verification using *check*, accounting the *Bounded* model checking technique.

Being SAT-based [43], the Alloy Analyzer tries to search for a *lasso trace* instance that satisfies the formula $M \models \psi_f$ over the *check* command, it performs the latter refutation ($(M \wedge \neg\psi_p)$ by applying *De Morgan's laws*), yielding a counter-example if the latter is satisfied. This technique is called *Proof by Refutation*: M entails ψ_p, denoted by $M \models \psi_f$, *if and only if* $(M \wedge \neg\psi_p)$ is unsatisfiable, reducing validity to unsatisfiability.

```
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 20: 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 steps is set to *10* by default, although this may be adjusted using the keyword $steps$, alongside with the bounded scopes specification.

The following run example makes use of both scopes and steps definitions. Moreover, the formula specified in the run states the expected behavioural representation of an *Eulerian* circuit. The $eventually$ (Figure 22) operator is considered to specify the desired final state, where every edge is visited and the *Euler* node finishes where it started.

```
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.

This particular run yields no instance, so its corresponding formula $M \wedge \psi_p$ does not hold, since imposing *5* steps is not sufficient to provide an instance where the desired state, defined by the latter formula, is reached.

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 [5] .

Alloy makes use of the relational logic (Figure 13) to ensure the well-formedness of the system, which falls short when it comes to supplying support over temporal behaviour. Thereby, Alloy includes temporal connectives from FOLTL semantics, that acknowledges the system's states along the trace evolution [43, 31, 10].

Ensuring the correctness of a system's behaviour relies on verifying properties regarding the latter's *Safety* and *Liveness*. Its verification technique is motivated by distinct approaches [38]. The former requires an invariance argument, where the latter requires a well-foundedness argument to prove its system satisfiability [3].

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

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

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

```
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.

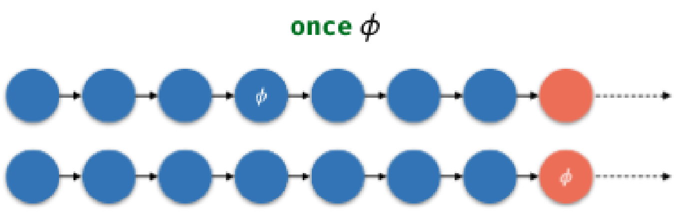


Figure 21: *once* behavioural representation.

Whereas, a *liveness property* expresses that "something good will happen", implying the eventual occurrence of a state during the course of the system's execution [40].

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

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

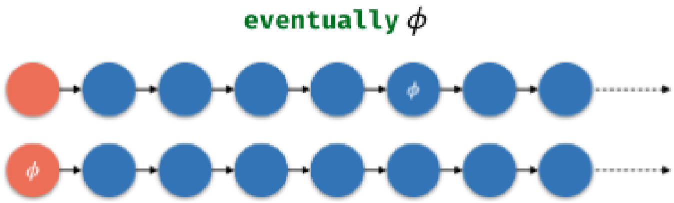


Figure 22: *eventually* behavioural representation.

The above-mentioned properties make use of *2* relevant over time quantifiers. The *once* operator expresses the past validation of a given formula. The *eventually* operator expresses an existence quantifier, imposing the formula verification somewhere along the trace evolution [9].

Moreover, the liveness property states the expected behavioural representation of an *Eulerian* circuit, where the *eventually* (Figure 22) operator is considered to specify the desired final state. However, its verification, using *check liveness_euler*, 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 handled by adding fairness to the trace assessment.

Liveness properties evaluation process reasons about states that eventually might reach the wanted scenario, it is advisable to consider *fairness* constraints, intentionally specified to rule out infinite traces with unrealistic behaviour, such as system stuttering [5]. Briefly, a *fairness* constraint imposes fairly considerations on the system's trace evolution [65].

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

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

The latter imposes a fairness constraint regarding the trace evolution, ensuring that trace stuttering is not possible as long as the *Euler* node has some adjacent node that is not been visited yet. As expected, by instructing the Analyzer to check the asserted property (*check liveness_euler*), it will return no counterexample.

```
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.

3.3.2 *Alloy Analyzer*

Normally, the specification process upon abstracting the system as a conventional model is carried out interactively, since considerations about model validation must be preceded by the model specification. The already mentioned analysis techniques, run and $check$ analysis commands respectively, instruct the Analyzer to check and provide analysis over their implicit formula nature.

Moreover, the Analyzer is capable of providing model instances concerning the formula evaluation, depicted graphically as graph-like structures, through the usage of the Alloy *Visualizer*. The latter allows instance interaction 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 $steps$ explicit 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 regarding the latter, consequently producing a visual graphical (Figure 23) through the usage of the Visualizer.

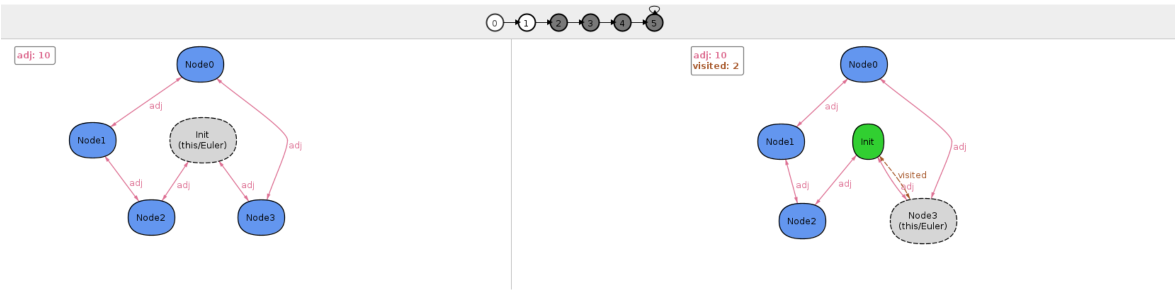


Figure 23: Partial graphical view of the *two* initial states. The *Euler* node starts in the *Init* node, and then moves towards an adjacent node.

The Visualizer furnishes a toolbar with multiple configuration buttons, allowing interactive customization over alternative instances [9]. Traces can also be interactively overviewed through transition buttons ($\rightarrow$ and $\leftarrow$), enabling forward and backward trace navigation.

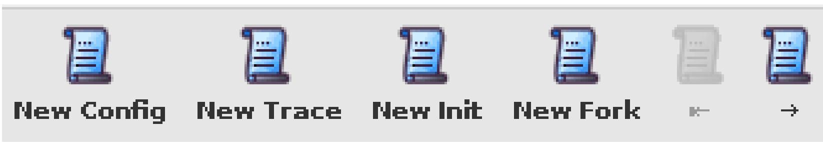


Figure 24: Alloy Visualizer toolbar.

The New Config instructs the Analyzer to provide a new trace configuration, where immutable model expressions (sets and relations) are depicted with new values. Consequently, the Visualizer will present an execution trace regarding the new model configuration. The New Trace instructs the Analyzer to present a new execution trace, regarding the same model configuration. The New Init requests for a new trace representation, 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 new transition post-state. The latter could differ on the result of a given event, or it can display the outcome of a different event [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 looping state is reached.

Concerning performance in the evaluation over infinite traces, Alloy considers the Bounded Model Checking technique as the first 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, due to the SAT time-bounded nature. However, it still confers great functionality, as infinite traces can often be represented as finite traces, making verification within small scopes possible [34].

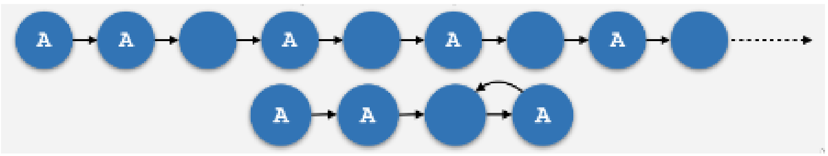


Figure 25: Some infinite traces can be represented by finite *lasso* traces.

Oppositely, the Analyzer can be instructed to perform *Unbounded Model Checking* towards achieving consistency over model verification, without bounding traces upfront [31, 9]. This is feasible as the state space is finite due to the Alloy analysis way of bounding signatures. Consequently, it constrains infinite traces as periodic *lasso* traces (Figure 25).

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.

4

STATE OF THE ART

4.1 RELATED WORK

This section intends to present previous developed work regarding the main concepts on what this dissertation rests. The subsection 4.1.1 aims to provide a comprehensive overview over works that attended to prevent security issues related to the deployment of robotic systems, using Robot Operating System as its application enhancer. It is then followed by the subsection 4.1.2 that concerns about previous work addressing property verification and model checking techniques over ROS applications.

4.1.1 *Security Overview*

The literature concerning the network security enhancment that Robot Operating System 2 furnishes, by offering the SROS2 toolset, is quite limited. Most of the existing work is on the exploration of the former version of ROS in terms of port exposure, contextualized in the approach considered to protect the system network.

Many researches were made regarding this issue that ROS faces, one in particular that explored the IPv4 address space of the Internet for instances of ROS, named *Scanning the Internet for ROS: A View of Security in Robotics Research* [20], with the goal of identifying ROS vulnerable hosts, mostly master nodes since they provide information about their related topics and node's parameters, mainly by port scanning, so that developers could be aware of the possibility of exposure of their robots.

Following the need of supplying security assurance over ROS applications, several approaches were presented. A study that is rather relevant because of the similarity between their proposal and the one that SROS2 has to offer is the one presented on the *Application-level security for ROS-based applications* [23]. 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. Even though encryption and authentication measures are concerned, the protected network is still perceived from the "outside", meaning that security attacks, such as denial of service, still persist which cannot be handled on the application level alone. Secure Robot Operating System (SROS) [69] was initially developed as an experimental tool (later evolved to SROS2 as a supporting tool for ROS2), which supports TLS for all socket transport, node restrictions and chains of trust, guaranteeing publishers authorization when it comes to publish to a specific topic. Another worth-mentioned tool is Rosbridge [18], which provides a WebSocket interface to ROS and corresponding server to allow interaction between applications and ROS nodes, by using TLS as support and also access control over topics and API calls.

The present works addressing ROS security methods tend to concern solutions to prevent vulnerabilites and issues that might compromise robotic applications deployed, while considering performance as priority. In terms of applying formal methods to verify properties regarding the domain of ROS2 and ROS2 security as models, there are minimal existent works. Despite this, the following section consists of several articles proposed to validate robotic systems, using formal methods as core.

4.1.2 *Analysis and Verification*

Static analysis over ROS represents a major contribution to this domain, in which researchers aim to tackle issues arised from miss configurations or code inconsistencies. The noteworthy *HAROS* framework [60] holds great value thanks to its contribution on improving ROS's software quality. *HAROS* makes use of several analysis techniques to exert quality evaluation of ROS software, followed by ways of feedbacking inconsistencies using predefined code metrics. As this framework seeks to be flexible when it comes to adding functionality, further static analysis works improvements have been proposed as plugins. In both 59 and 58, it is presented additional functionality to the framework, through applying architectural considerations over metamodel designing, where the latter supports the former by supplying property-based specifications. These techniques confers great help back to developers, since static analysis offers advantageable usage over raw review of software code.

The literature concerning property verification over model checking tools is quite extensive. Regarding ROS applications, some approaches were presented that mainly focused on modelling the ROS node-communication, while real-time properties were also considered as support to the target language. In 32, *UPPAAL* model checker is used to model ROS applications, supported by a concrete robot example, that is followed by techniques to verify properties regarding its behaviour. In 10 is presented a notable proposal, where *Electrum* [43], the former version of the current Alloy Analyzer, is used as an additional plug-in to the already mentioned *HAROS* framework. Through ROS launch configurations, the plugin automatically generated models using Electrum and performs verification over these models, to then feedback issues related to their ROS system behaviour.

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 2, 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 systems, in 42, DDS in ROS2 is formalized as a timed automata, consequently followed by model verification over property-checking. These works conceive value concepts and procedures useful for this dissertation contextualization.

A few studies on robotics should be recognized in which techniques such as model-checking were performed. Despite the fact that they do not address the domain of ROS, they nonetheless give helpful background for the robotics research over formal approaches. A case study, mentioned in 51, presents a novel approach over systems that require static analysis based on software assumptions and proper analysis within its environment usage, where user-interaction comes in hand. Concerning this novel idea, a medical system is concerned as a case study, where multiple safety-based considerations are expected, as well as, an end-to-end critical property that must be satisfied over the entire analysis course. Another notable work regarding former analysis using Alloy specification language is presented in 44, 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 these studies presents favorable outcomes, their focus lies on a particular area of study. As a result, they lack on providing solutions to a vast majority of situations.

4.2 CURRENT WORK

BIBLIOGRAPHY

[1] Robot Operating System 2. Sros2. https://github.com/ros2/sros2, 2021.

[2] 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.

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

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

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

[6] 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.

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

[11] 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.

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

[13] 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.

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

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

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

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

[18] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504. Springer, 2017.

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

[20] 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.

[21] 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.

[22] 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.

[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.

[24] 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.

[25] 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.

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

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

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

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

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

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

[32] 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.

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

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

[35] Daniel Jackson. Alloy: a language and tool for exploring software designs. 62:66–76, 2019.

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

[37] 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.

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

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

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

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

[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, 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.

[59] 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.

[60] 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.

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

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

[63] 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.

[64] 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.

[65] Thomas Wahl. Fairness and liveness.

[66] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

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

[68] 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.

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

[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.