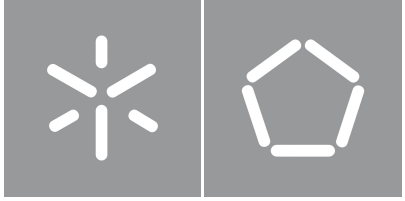
University of Minho
School of Engineering

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

december 2022



University of Minho
School of Engineering

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

Masters Dissertation
Integrated Master's in Informatics Engineering

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

december 2022

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

First, I would like to thank my family for being comprehensive along my life path and fully supportive of my decisions. Also, to my dearest Catarina, the girl who lift me up whenever I needed the most, especially when all the work seemed purposeless and I was at my lowest of motivations, I am truly grateful to have you.

To the friends and colleagues that I came across during these last years of my academic life, thank you for helping through these tough years. Of these, I need to especially thank to Morais, José Pedro and Pedro for being present, sharing thoughts and knowledge within a friendly environment. Also to Ricardo, Diogo and Tiago, thank you for standing by my side for years.

Last, but not least, I would like to thank to my supervisors, Alcino and André, for being totally supportive with me. Also, to Nuno, who helped me during the early phases of this dissertation. I am delighted to have had the chance to acquire a small portion of your expertise, within a great spirit and environment.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

FCT Fundação para a Ciência e a Tecnologia

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.

University of Minho, Braga, december 2022

Luís Mário Macedo Ribeiro

Abstract

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

Following the current growth of cyber-physical systems, as well as their usage in various technology domains, the development of software applications is becoming more demanding due to the complexity behind the integration of complementing services, beyond those provided by the operating system.

One of the most popular open-source software platforms for building robotic systems is the **Robot Operating System (ROS)** [53] middleware, where highly configurable robots are usually built by composing third-party modules. **Robot Operating System 2 (ROS2)** is implemented using the **Data Distribution Service (DDS)** [49] communication protocol. ROS2 implicitly makes use of the DDS-Security artefacts through the **Secure Robot Operating System 2 (SROS2)** security toolset.

The present study focus on detecting security problems in ROS2 networks, in which it is intended to verify, through formal techniques, security properties. However, security is a very broad subject, so this study focuses on a particular security property to show the viability of the proposed technique, namely **Observational Determinism (OD)**.

This dissertation introduces a software tool, named **Security Verification in ROS (svROS)**, that provides multiple functionalities to support this type of security analysis using Alloy [32], a formal specification language and analysis tool.

Keywords Robotics, ROS2, SROS2, Security Properties, Observational Determinism, Software Verification, Alloy

Resumo

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

Como resultado do atual crescimento dos sistemas ciber-físicos, nomeadamente sistemas robóticos, bem como a sua utilização em vários domínios tecnológicos, o desenvolvimento de aplicações tem vindo a ficar mais exigente, em particular devido à complexidade da integração dos serviços necessários, tipicamente não fornecidos pelo sistema operativo.

Uma das plataformas considerada como *standard* no que toca ao desenvolvimento de sistemas robóticos é o *middleware* **ROS** [53], onde robôs altamente configuráveis são construídos atráves da composição modular de *software* externo, oferencedo características como flexibilidade e interoperabilidade aos sistemas integrados. O **ROS2** implementa um protocolo de comunicação, de nome **DDS** [49], que, para além de garantir serviços de comunicação, implementa a especificação DDS-Security, que oferece diferentes métodos de adoção de segurança, através de uma metodologia de *plugins*. Através do uso desta especificação, juntamente com o uso do *toolset* **SROS2**, é possível configurar o ROS2 de forma a propocionar um ambiente seguro às aplicações integradas.

O presente trabalho foca-se no estudo e deteção de problemas de segurança em topologias ROS2, através da verificação formal de propriedades de segurança. No entanto, a segurança é um assunto extenso, pelo que o foco de interesse nesta tese é numa propriedade particular de segurança para mostrar a viabilidade da presente técnica, de nome **OD**.

Esta dissertação introduz a uma ferramenta de verificação de nome **svROS**, que contempla múltiplas funcionalidades para suportar este tipo de análise usando Alloy [32], uma linguagem de especificação formal e respectiva ferramenta de análise.

Palavras-chave Robótica, ROS2, SROS2, Propriedades de Segurança, Determinismo Observacional, Verificação de Software, Alloy

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AES Advanced Encryption Standard. 31

CA Certificate Authority. 32

DCPS Data-Centric Publish Subscribe. 23

DDS Data Distribution Service. iv, v, 1, 21

DSL Domain Specific Language. 58

FOLTL First Order Linear Temporal Logic. 16

GNI Generalised Non Interference. 45

HAROS High-Assurance Robot Operating System. 36, 56

LTL Linear Temporal Logic. 2, 5, 43

NI Non Interference. 44

OD Observational Determinism. iv, v, 2, 40, 45

OID Object Identifier. 32

OMG Object Management Group. 23

PKI Public Key Infrastructure. 29

rcl ROS Client Library. 24

rmw ROS Middleware Interface. 24

ROS Robot Operating System. iv, v, 1, 21

ROS2 Robot Operating System 2. iv, v, 1, 21

SPI Service Plugin Interface. 29

SROS Secure Robot Operating System. 35

SROS2 Secure Robot Operating System 2. iv, v, 21, 31

SSL Secure Sockets Layer. 31

svROS Security Verification in ROS. iv, v, 3, 53

TLS Transport Layer Security. 31

Chapter 1

Introduction

Automation is increasingly being incorporated into the industrial world, through the use of flexible tools to assist in the most various scenarios, as this brings efficiency and accuracy to the industry's processes. Robotics is already the key driver of competitiveness and flexibility in large scale manufacturing industries. Due to the continuous growth of technology in these different domains, robots can be used in a wide range of applications [47] because their usage increases productivity, safety, and manufacturing production work back to developed countries.

Despite the advances in technology, dealing with hardware-level applications has become highly impractical as systems' complexity increases. Therefore, developing and writing software code for robot applications is becoming increasingly demanding because multiple aspects must be properly considered [52].

Since robots are essentially integrated as distributed systems with separate components, the need to connect different hardware and software modules raises interoperability and communication issues. To solve these issues, modular architectures, based on message-passing communication patterns, are continually emerging as the middleware layer of architectures. Their primary focus is to offer services to the application layer, consequently easing the development cost while providing interoperability and communication facilities [47, 45].

The **Robot Operating System (ROS)** was created by a collaborative open-source community to contribute to the advancement of robotics, in particular with the aim of helping to build robotic applications easily [26]. It provides locomotion, manipulation, navigation, and recognition tasks via third-party software libraries and tools. Concerning the wide range of robotics hardware and software, ROS was designed to be flexible, enabling the easy integration of external components. However, performance and scalability issues arise due to its middleware specification [52].

These issues led to the creation of **Robot Operating System 2 (ROS2)** [54], which was developed using the **Data Distribution Service (DDS)** specification protocol as its communication middleware.

Issues concerning system integration and scalability are mitigated by DDS various implementations owing to the several transport configurations provided, making it suitable for real-time distributed systems. DDS also provides a security specification, called DDS-Security. ROS2 makes use of this specification to provide security guarantees to robotic systems, through policies of access [5].

Security in distributed systems does not exclusively rely on entities' control of access. Despite the fact that policies restrict entity data access, vulnerabilities and private-data leakage can still arise as a result of misconfigurations. Avoidable security behaviour can be addressed the same way as a common behaviour, namely through property specification and verification. Security properties, however, are not entirely expressed by conventional temporal logic properties, as they require untrusted inputs and the resulting outcomes to be compared, which can only be achieved by comparing different execution traces. Hyperproperties [17] cover such conditions, as they denote a predicate on sets of executions, adding expressiveness to temporal logic.

Because of the widespread use of robotic systems, software verification through the use of formal methods is necessary to prevent potentially catastrophic consequences, mainly related to security matters, as safety guards are gradually implemented in the software domain [70]. Within this context, the Alloy [36, 42] framework enables the behavioural representation of systems with rich configurations owing 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, are more reliable than traditional testing.

Hyperproperties, however, cannot be directly expressed in Alloy, or any other conventional model-checking framework, as they require more than a single execution trace to be verified upon. Accordingly, to perform verification of properties in multiple execution traces simultaneously, the system model must be replicated to allow their simulation in a single system execution. This technique is known as Self-Composition [4], which allows some hyperproperties to be verified through conventional model-checking frameworks.

The overall goal of this thesis is to apply formal methods to the security verification of ROS2 applications. Given an application architectural and communication topology, along with its access policies, an Alloy model is inferred, whose structure accounts for both self-composition and synchronisation between executions. **Observational Determinism (OD)**, the only hyperproperty addressed in this thesis, is then verified using Alloy's model-checking capabilities.

1.1 Objectives and Contributions

The first goal of this thesis is to survey the concepts around ROS2, contextualising the evolution behind this framework towards achieving security, in which the former version of ROS lacked due to the focus on flexibility. Since ROS2 has been developed over the DDS framework as its communication middleware, DDS must be properly understood before considering the security aspects.

The second goal is to extend a previously proposed [12] formalisation of ROS applications in Alloy [32, 42], to also take into consideration the security configuration defined with SROS2.

The final goal of this dissertation is to develop a novel technique to automatically verify security properties of ROS2 applications using the Alloy framework. In particular, the goal is to extend the ROS2 Alloy models to properly address self-composition, to enable the verification of OD.

With this goal in mind, a prototype verification tool was developed in this work – **Security Verification in ROS (svROS)** – that prioritises the early detection of security configuration issues on ROS2 applications. It offers a multitude of functionalities that help a common developer to fetch source code and create default templates, including a SROS2 policies file, whose information is then used to perform verification of OD in an application, without the need to launch it.

One of the contributions of this thesis is to present to the ROS community an overview on how ROS systems can be indirectly exploited if security behavioural properties such as OD are not ensured. As far as we are aware, hyperproperties have not yet been addressed in Alloy, so another contribution is the first proposal of a technique to verify OD in Alloy.

1.2 Document Structure

The reminder of this document is divided into *5* different chapters, where the first *2* regard the state of the art (Chapters 2 and 3) followed by the work developed (Chapters 4 and 5), alongside with its evaluation (Chapter 6) and future considerations (Chapter 7).

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

Chapter 3 introduces all the concepts related to ROS, and its evolution as robotic development framework towards achieving system security. Subsequently, the DDS and its security specification architecture are mentioned and studied. In Section 3.3, some current works that fit the domain of software verification in robotic systems are also overviewed.

Chapter 4 gives a proper understand to the reader about hyperproperties and how relevant these are to the study of verification within distributed systems. It is supported by a case-study ROS2 application, with focus on a highly relevant security property, named OD.

Chapter 5 presents svROS, a verification tool that was developed to perform verification of OD in ROS2 applications. It introduces the reader with all its capabilities for code extracting, structural parsing, and behavioural specification.

Chapter 6 evaluates the usability and performance of this tool. For the sake of illustration, it uses the case-study of Chapter 4, presenting different time-measurement evaluations.

Finally, the Chapter 7 draws the conclusions and some considerations about the future work are outlined.

Chapter 2

The Alloy Specification and Analysis Framework

The increasing usage of robotics in safety-critical systems requires ensuring the proper correctness of both software and hardware, as failures often lead to fatal consequences, namely regarding the security domain. Thus, the use of formal methods and verification techniques, especially in systems that are highly reliant on flexibility and reliability, is recommended to avoid security-critical faults [15]. Software frameworks designed for this purpose must provide methods to perform structural design over systems with rich structures, abstracting their behaviour as a conventional formal model. Additionally, these frameworks must support features to enable automatic analysis, in which property verification over these design models is used as technique.

The *Alloy Framework* [36] fits within this context, as it provides a declarative relation-based language used for software modelling, complemented with extensive tool support for analysis over these models [32]. The language combination of both *Relational* and **Linear Temporal Logic (LTL)** enables modelling of both systems with rich structures and complex behaviour. To address the correctness of the specified model, Alloy supports model-checking for these logic languages, where a property is exhaustively checked over a model M when performing verification [42].

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

This chapter goes through these principles in further depth, to give the reader a proper understanding on how Alloy is structured, supported by an example related to *Eulerian* Circuits. Since the system analysis relies on model-checking techniques, the following section starts by briefly introducing this topic.

2.1 Model-checking

Performing software testing has been regarded as the established assessment procedure to check functional and non-functional specifications. The conventional approach for software validation is based on testing the system with different inputs, to achieve quality assurance over several intended specifications [6, 9]. As this technique demands exhaustive evaluation over pre-selected test data, it is commonly applied with automated tools, since manual testing is time-consuming and prone to errors [16, 29].

Model-checking is a technique with the purpose of verifying temporal properties over a finite-state model of a system. Additionally, it enables *model-based testing* by automatically interpreting counterexamples as test cases, resulting in significantly greater degrees of coverage than conventional testing [29, 6]. This technique is increasingly used because of its importance as an early phase verification technique when developing systems [42].

Model-checking provides a highly automatic verification procedure, unlike other techniques, such as theorem provers, which are based on deductive reasoning. However, the large number of states in realistic models often causes an increase in complexity during verification. This is referred as the *state explosion problem*, which occurs when model-checking cannot handle the size of the state space [15, 14]. Yet, this can be mitigated using *bounded* model-checking techniques, which limit the exploration of finite-state models to a given depth.

Model-checking techniques usually require properties to be specified with some sort of *Temporal Logic*. Thus, the goal is to check whether a temporal logic formulas hold in a finite-state model of a system [34, 62].

A *Transition System* is defined as a graph-based representation of a system dynamics, that confers additional representation over the mathematical graph structure. For model-checking purposes, the transition system must be equipped with a labelling function that maps each state to the set of *atomic propositions* that are true in that state. These propositions describe the values of the system variables for each state [48, 62].

A transition system with such labelling function essentially corresponds to a *Kripke Structure* [48]. Conceptually, a *Kripke Structure* corresponds to a model M with the following tuple structure $M = (S, I, R, p, L)$, where: S is a finite set of states; I represents a set of initial states, so naturally I is a subset of S ($I \subseteq S$); R defines the *transition relation* as it accounts for the transitions between states ($R \subseteq S \times S$); L is an *interpretation* that defines the labelling function, namely it assigns each state with a set of valid atomic propositions $L(s)$ that are valid in it, drawn from the domain p ($L(s) \subseteq p$).

Model-checking is a *model-based* technique which in verification focuses on the concept of property satisfaction. Therefore, a model-checker must be capable of checking if the model M satisfies a desirable property, expressed as a temporal logic formula ψ: $M \models \psi$ [34, 48].

The semantics of temporal logic relies on how the latter addresses time as an evolving entity towards state verification. Notably, temporal logic is qualified as either *linear-time* or *branching-time* [34]. In the former approach, time is perceived as a 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 nondeterministic considerations about the system evolution. The choice on the logic semantics depends on the system properties to be analysed [48] and entails different model-checking algorithms [34]; nevertheless, Alloy uses linear-time logic to quantify over temporal formulas.

2.2 Modelling

The Alloy framework presents itself as a formal modelling language for both structural and temporal behaviour design. Formerly, Alloy was inherently static [42], meaning that it only excelled in structural design, with a specification language based on first-order relational logic. The analysis process relied on a bounded model-finding technique, with no support for temporal behaviour. Notwithstanding, the latest release of Alloy[1] confers the ability to properly deal with expressive temporal properties, as well as trace evaluation over time [32].

2.2.1 Structural Modelling

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

The Alloy structural definition relies on a relational approach to connect the elements of a system, where all the structures are modelled with relations. Unary relations, commonly known as sets, are denoted as *signatures* in Alloy. Signatures are inhabited by a set of *atoms*, drawn from a finite universe of discourse. Atoms are perceived as the lowest-grain elements, with no particular semantics attached. A signature, declared by the keyword `sig`, might include multiple *field* declarations enclosed between braces,

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

to declare relations between the signature's atoms and other signatures' atoms. Fields are inhabited by tuples of atoms from the universe, that must have the same arity.

A signature can either be a top-level signature or a subset of another signature. Signature hierarchy is specified through disjoint extensions (`extends`) or by set inclusion (`in`). The `abstract` keyword declares a signature that contains no atoms beyond those within its extensions.

Both signatures and fields can be specified with a multiplicity constraint. In the former case it constrains the number of signature atoms and is commonly used to express singleton sets, with the keywords `one sig`. Field declarations, however, usually use a wider range of multiplicities to restrict the content of the relations. In addition to these implicit multiplicity constraints, system assumptions can be defined with axioms, expressed as *facts*, where multiple constraints can be incorporated.

Throughout the sections that follow, we present an illustrative example from graph theory related to *Eulerian circuits*. This example is used to contextualise both the modelling and the verification process in Alloy.

The structural model that captures a graph structure must be provided beforehand. Eulerian circuits are a refinement of such model, as these meet several behaviour constraints over the graph definition.

At an abstract level, graphs can be represented as a set of nodes, connected together with an adjacency relation, with no need to address edges as a separate structure declaration.

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

Listing 2.1: Graph representation. The **sig** keyword is followed by the corresponding *Node* signature declaration.

The *Node* signature is defined by a static set of node atoms, that combined denotes the finite universe of discourse. Fields are enclosed between braces. The *adj* relation models 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. The *visited* relation is introduced in the next section.

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

```
one sig Init extends Node {}
```

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

Listing 2.2: *Node* signature hierarchy. As *Node* is not preceded by the **abstract** keyword, its atoms do not solely belong to the *Init* signature.

Both these signatures are preceded by the one operator, imposing a multiplicity constraint over each signature declaration. Setting the multiplicity to one means that each model instance must have precisely one *Init* atom. The *Euler* signature is introduced in the next section.

Modelling constraints can be specified by making use of the fact declaration. The formula specified in each fact declaration denotes a model axiom, that serves as a premise for further reasoning.

Eulerian circuit An Eulerian circuit denotes a trail within a finite graph, with each graph edge being visited precisely once and the trail starts and ends on the same node . Thus, the graph must be connected, where each node must be reachable from each other, and not-directed. Moreover, it is assumed that the graph has no self-loops.

```
fact graph_restrictions {
    adj = ~adj // Not directed
    no iden & adj // No self-loops
    all x : Node | x->Node in ^adj // Connected
}
```

Listing 2.3: Graph restrictions through an axiom specification.

These constraints make use of some Alloy operators, which can either be identified as set-theory operators or as relational operators (Figure 1).

Regarding set-theory, this fact uses set intersection (denoted by &) and Cartesian-product (denoted by ->). The first-order logic quantifiers can also be used in Alloy and they may allow a more understandable way of specifying certain constrains. The last constraint makes use of the universal quantifier all , with x : Node denoting the range of the quantifier, in this case every member x of Node, and the desired restriction specified after the | separator.

This fact also uses some relational operators. The ~adj denotes the converse relation of *adj*. The transitive closure operator (^), defined in Figure 2, yields the smallest transitive ^adj relation containing all the pairs of tuples, reachable in one or more steps, through the implicit use of the set composition operator (.). The reflexive transitive closure (*) is defined as *adj = ^adj + iden .

In addition to having constraints defined in explicit axioms, structural constraints can also be implicitly

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

iden	id
Φ + Ψ	$\Phi \cup \Psi$
Φ & Ψ	$\Phi \cap \Psi$
Φ - Ψ	$\Phi \setminus \Psi$
Φ -> Ψ	$\Phi \times \Psi$
Φ . Ψ	Φ . Ψ
A <: Φ	$A \triangleleft \Psi$
Φ :> A	$\Phi \triangleright A$
~ Φ	Φ°
^ Φ	Φ^{+}
* Φ	$\Phi^{\star}$
$\{ x : A \mid \phi \}$	$\{x \mid x \in A \wedge \phi\}$

Figure 1: Relational Logic Syntax. Left-handed table depicts atomic formulas, whereas the right-handed one depicts expressions.

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

Figure 2: The transitive closure operator.

defined in a signature declaration. These are often referred as signature facts, universally quantified over a signature's set [2]. For example, the second constraint, stating that the graph contains no self-loops, can be declared as a signature fact as follows.

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

Listing 2.4: The no self-loop constraint defined as a signature fact. Even though `adj` is introduced as a binary relation between `Node`s, in a signature fact it is implicitly projected to each `Node`. The keyword **this** denotes the implicitly universally quantified variable.

2.2.2 Behavioural Modelling

A model behavioural specification expresses what is intended to happen during state transitions over the valid traces of a system. A *trace* is represented as an infinite chain of states that completely describes a system's potential behaviour. Valid traces are constrained by the explicit specification of axioms [30].

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

Field *visited* represents a mutable relation containing the visited edges, meaning that its evaluation may change during the course of the trace's evolution, as opposed to static ones. The *Euler* node, which captures the current position while traversing the graph, is also accounted as variable because it can be inhabited by different atoms in different states.

The most common idiom to specify the system's behaviour is to use a temporal formula that restricts valid transitions to a set of events.

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. An effect specifies how the system evolves to the next state, specifying a valid outcome of the event. The effect must refer to the mutable variables of the model through the usage of the $'$ operator, that evaluates an expression in the next state. Alternatively, the temporal operator *after* can be used to ensure the truth of a formula in the next state.

Events are conveniently specified in separate *predicates*. These are declared using the keyword `pred`, which express boolean formulas that only hold in that state when invoked. However, as we use the $'$ operator to evaluate the next state s', while also evaluating the current state s, we are indirectly evaluating the whole transition from s to s'; i.e, an event.

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

Listing 2.5: The event of traversing an edge. The first guard requires the existence of some node n, that is adjacent to the *Euler* node. The other guard ensures that the edge was not visited before. It is followed by two effects, the first updating the *visited* relation to include the latest visited edge, and the second setting the new node for *Euler*.

Unexpected behaviour can occur if no constraints are set on how the system should evolve a mutable structure. Constraints imposed on mutable structures that should remain unchanged in the next state are referred as *frame conditions*, a formula with primes stating the no change effect [10]. It is also advisable to consider the nothing changes possibility in the specification of the trace evolution.

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

Listing 2.6: The stutter event. It captures such no effect event and consists only of frame conditions.

Every conceivable trace of the system must be an interleaving of these events, starting in a well-defined initial state.

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

Listing 2.7: The initial state. It denotes the valid conditions in the first execution state, namely that 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. A system's permissible behaviour can be specified by restricting the set of valid events. This can be done with the following axiom.

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

Listing 2.8: Trace constraint through the use of an axiom.

The *always* temporal operator (Figure 3) expresses a universal quantifier over time, imposing a constraint that should be valid in every state of a trace. In the fact `traces`, this operator is followed by the desired constraint, namely that one of the two events must always occur. This fact also invokes the `init` predicate to ensure the proper initialisation of the system.

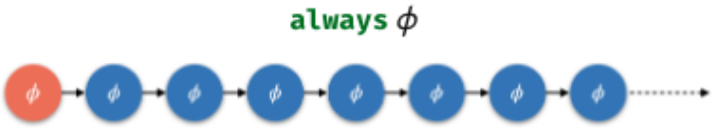


Figure 3: $always$ informal semantics.

This behavioural modelling technique implicitly defines a *Kripke* structure. The set of initial states (I) is imposed by predicate `init`. The transition relation R accounts for both predicates `traverse` and `stutter`, which denote the transitions between states along the trace. Each state s ($s \subseteq S$) is defined as a possible valuation to the declared signatures and fields.

2.3 Analysis

The latest version of Alloy adds linear temporal logic to first-order logic, thus allowing both temporal and relational operators when specifying the desired properties [42]. This section aims to explain how Alloy conducts analysis of such properties, with further explanations over the corresponding analysis commands, along with an overview on the interactive exploration process of the Alloy Analyzer.

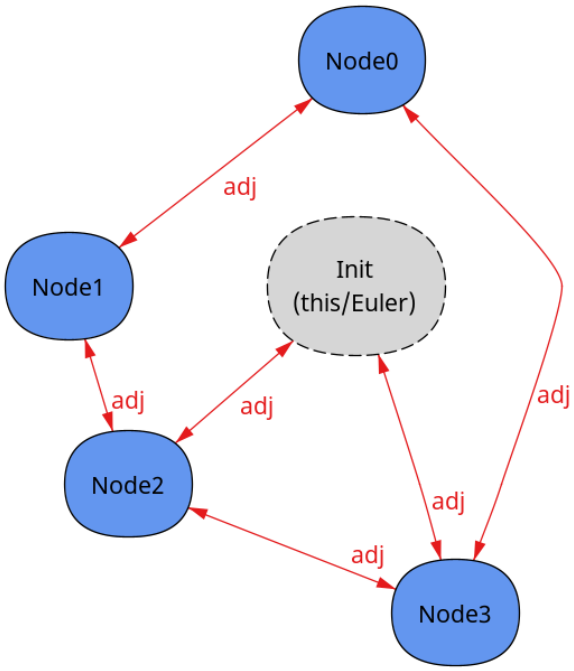


Figure 4: Acceptable configuration and initial state of Eulerian graph.

2.3.1 Analysis Commands

The Alloy language supports two analysis commands, `run` and `check`. A temporal logic formula should be enclosed between braces inside a command. Briefly speaking, a `run` command instructs the model-checker to present an example that satisfies the enclosed formula ψ_f as well as the model specified in the remaining facts (denoted by M). This means that the formula $M \wedge \psi_f$ is expected to hold in the returned instance. The consistency of the facts and signatures is consequently also verified since the model declaration M is also considered.

After modelling the structure and its constraints, it is advisable to run the model to ensure that no unwanted model structural scenarios are presented. Figure 4 depicts an instance of an acceptable configuration for the initial state of an Eulerian graph with `5 Node` atoms.

Alloy makes use of the `check` command to perform model-checking of an `assertion`. The verification of assertion ψ_f is done by checking that it holds in the model within the defined scopes, that is it checks whether $M \models \psi_f$ is valid.

Because of the undecidability problem of first-order logic, automatically proving formulas is not possible [62]. To ensure decidability, the Alloy Analyzer performs analysis with *scopes* assigned to each signature declaration. By default, if no scope is provided, the model-checker considers at most three atoms for each top-level signature.

Since scopes impose a limit on the state space explored by the Alloy Analyzer, if no instance is returned, the formula cannot be considered inconsistent (in *run* commands) nor valid (in *check* commands), since a bigger scope could allow the command to return an instance. However, due to the small scope hypothesis [36] it is likely that most commands can be satisfied with small scopes.

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

Figure 5: Bounded model-checking of formula ϕ.

Alloy uses bounded model-checking as the default analysis technique. The number of considered steps is set to ten by default, although this may be adjusted using the keyword `steps`, along with the desired scope specification (see Figure 5).

A check command `check` $\{\psi_f\}$ instructs the Alloy Analyzer to search for a *lasso trace* instance that refutes the judgement $M \models \psi_f$, by trying to find an instance that satisfies $(M \wedge \neg\psi_f)$. If such an instance is found, it is a counterexample to ψ_f. This technique is called *Proof by Refutation*: $M \models \psi_f$ *if and only if* $(M \wedge \neg\psi_f)$ is unsatisfiable, thus reducing validity to unsatisfiability. In the case of bounded model-checking, the command in Figure 5 is equivalent to checking the unsatisfiability of the run commands in Figure 6 in sequence, in order to return the shortest counterexample.

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

Figure 6: Reducing validity to unsatisfiability.

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

Listing 2.9: A run command that searches for an Eulerian circuit with at most 5 steps.

In this command, the *eventually* operator (Figure 7) expresses an existence quantifier over time, imposing that the formula holds somewhere along the trace [10]. Here, is used to specify the desired final state, where every edge is visited and the *Euler* node finishes where it starts.

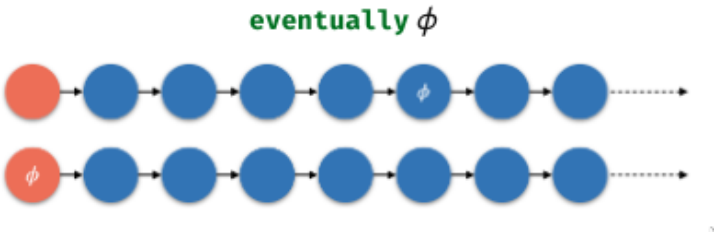


Figure 7: *eventually* informal semantics.

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

Checking Behavioural Properties

The verification process takes into account the formal specification of properties that are relevant to reason about the temporal behaviour of the system. To specify those properties, Alloy includes temporal connectives from **First Order Linear Temporal Logic (FOLTL)**, including past operators [42, 32, 12].

Ensuring the correctness of a system's behaviour relies on verifying properties regarding the latter's *Safety* and *Liveness*.

A *safety* property asserts that nothing bad should happen during the system execution. Consequently, if a trace that violates a safety property is found, it can be assumed that it has a bad prefix. As examples of safety properties, consider the following assertions. The first denotes an expectation about the evolution of the relation *visited*, namely that it grows over time.

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

Listing 2.10: *Safety Property*: The relation *visited* can only grow over time.

The second denotes that if a certain node is visited, the *Euler* node must have passed through it in the past. This assertion makes use of the *once* past temporal operator (Figure 8), which checks if a formula is valid somewhere in the past.

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

Listing 2.11: *Safety Property*: If a node is visited, then once *Euler* passed through it.

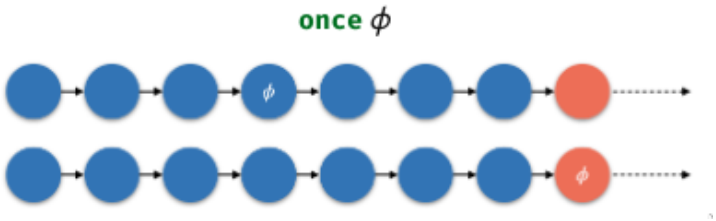


Figure 8: *once* informal semantics. The formula holds in the red state.

On the other hand, a *liveness property* expresses that something good will happen, implying the eventual occurrence of a state during the course of the system's execution [39]. This forces the system to evolve.

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

Listing 2.12: *Liveness Property*: Eventually all the graph will be visited and Euler is back in the inital node.

The above liveness property uses the *eventually* (Figure 7) operator to specify the desired final state. However, its verification yields a counterexample, as it is possible to always perform the stutter event; therefore, the expected behaviour never happens. This scenario results in an implausible infinite trace behaviour, which must be excluded by adding fairness constraints to the system specification.

To verify liveness properties, it is advisable to consider *fairness* constraints, intentionally specified to rule out infinite traces with unrealistic behaviour, such as permanent system stuttering [3]. Briefly, a *fairness* constraint imposes reasonable considerations on the system's trace evolution [63].

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

Listing 2.13: *Fairness predicate*: If *Euler* has an adjacent node, it will eventually visit it.

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

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

Listing 2.14: *Liveness* property accounting the fairness constraint.

Even though the fairness predicate denies the possibility of continuous stuttering, the liveness property still yields a counterexample, namely a graph with two nodes linked by an edge. The latter graph is not a proper Eulerian graph, since the necessary condition that all nodes in the graph must have an even degree, has not been imposed.

2.3.2 Alloy Analyzer

Besides analysis of commands, the Analyzer is capable of depicting instances graphically as graph-like structures, through the usage of the Alloy Visualizer. The latter allows instance navigation with multiple options. In addition, concerning user comprehension, the graphical depiction of these instances can be customised using *Themes*, through the Theme toolbar button.

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

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

Listing 2.15: A command that searches for an Eulerian circuit with at most 10 steps.

Finding concrete instances can be quite helpful throughout the modelling process. The run command presented above searches for instances with at most ten steps. The Analyzer executes the command and generates an instance, consequently producing a visual graphical representation (Figure 9) through the usage of the Visualizer.

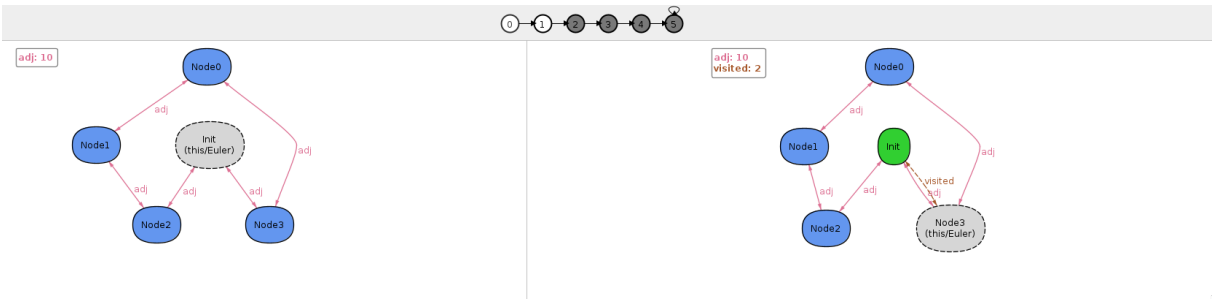


Figure 9: Partial graphical view of the two initial states of an Eulerian circuit. The *Euler* node starts in the *Init* node and then moves towards an adjacent node.

The Visualizer has a toolbar with multiple navigation buttons (Figure 10), allowing interactive exploration of alternative instances [10]. Traces can also be navigated through transition buttons (→ and ←), enabling forward and backward trace navigation.

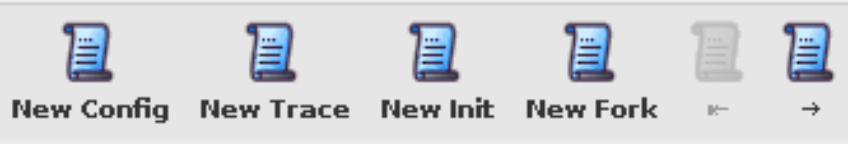


Figure 10: Alloy Visualizer toolbar.

The New Config instructs the Analyzer to provide a new trace configuration, where immutable model structures (sets and relations) have different values. The New Trace instructs the Analyzer to present any different execution trace for the same model configuration. The New Init requests a new trace where the initial state mutable structures are forced to have different values. At last, the New Fork allows alternative transition behaviour exploration over the same starting state, depicting a trace with the same prefix up to the current state but a different post-state. The latter can differ on the result of a given event, or it can display the outcome of a different event [10].

Additionally, the Alloy *Evaluator* [32] confers additional functionality to the Visualizer. It allows the user to ask the value of formulas and expressions against the existing model, to be evaluated in the current state [10].

Alloy uses the bounded model-checking technique as the default approach towards the model verification. Here, the corresponding formula is verified for all *lasso* traces of size up to a bounded number of steps. So, the verification is not complete. However, this still confers great assurance, in part because most infinite traces can be represented as finite *lasso* traces (Figure 11), thus making verification with a small scopes possible [35].

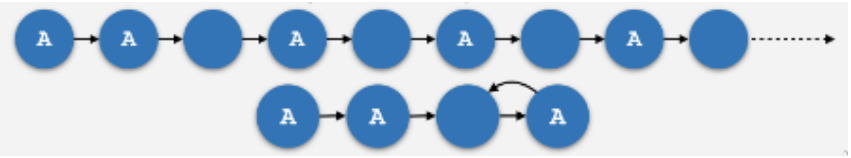


Figure 11: Some infinite traces can be represented by finite *lasso* traces.

The Analyzer can also be instructed to perform unbounded (or complete) model-checking, without bounding traces upfront [32, 10]. This is feasible as the state space is finite due to the scopes Alloy requires to bound signatures. To perform complete model-checking, an appropriate solver must be selected in the Options menu, and the `steps` scope must be specified using the following syntax: `for 1.. steps`.

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

Listing 2.16: A command that searches for an Eulerian circuit of any length.

Chapter 3

Software Development in ROS2

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

Due to the wide variety of robotic hardware in multiple domains, robotic software development is rather difficult [7]. The reuse of code is non-trivial, and therefore, large-scale development can become rather untenable. The **Robot Operating System (ROS)** presents itself as a middleware system, created to facilitate robotic system development in large scale.

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

The new version of ROS, **Robot Operating System 2 (ROS2)**, presents itself as a framework for developing robotic systems supported by the **Data Distribution Service (DDS)** standard. Consequently, ROS2 provides more functionality when compared to the former version of ROS, namely it provides the **Secure Robot Operating System 2 (SROS2)** toolset, to make available for ROS2 developers the security functionality that the DDS-Security specification [1] offers.

This chapter introduces necessary background information to ROS2 concepts, emphasising on how ROS2 implements security. First, it presents a detailed introduction to ROS concepts, as well as the evolution that ROS2 faced towards providing security to its deployed systems. In particular, DDS and its integration into ROS2 are properly contextualised.

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

3.1 Architecture Considerations

ROS was created by a collaborative open-source community, that has undergone rapid development [7] to contribute to the advancement of cyber physical systems. It was purposefully designed to be a development enhancer for the realm of robotic applications [26, 52].

Fundamentally, ROS is a middleware that provides a custom serialisation format, a custom transport protocol, as well as a custom central discovery mechanism. It represents 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 [13], allowing ROS applications to be built over several software modules, as independent computing processes called *nodes*. These are composed together to fulfil the deployment goal of the corresponding robot [45].

3.1.1 ROS Architecture

ROS architecture is based on a hybrid peer-to-peer implementation, where network communication is done with message-passing through a publish-subscribe pattern. The communication API relies on a stateless XML-encoded remote procedure protocol [67, 25].

This architecture (left-hand side of Figure 12) approaches communication through a centralisation perspective. It relies on the explicit implementation of a *Master node*, that controls every aspect of the communication establishment.

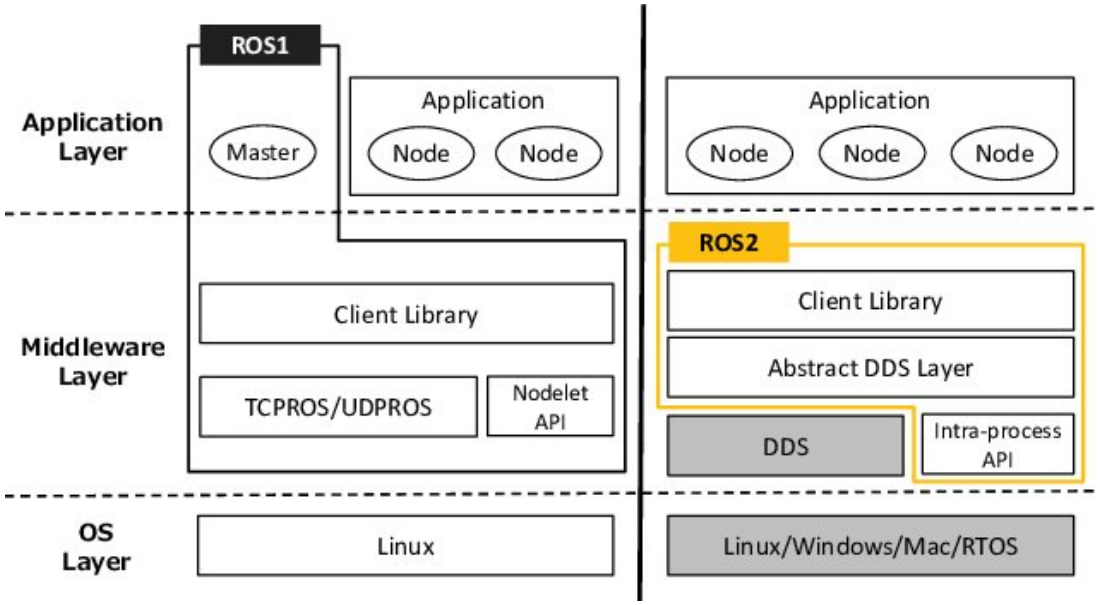


Figure 12: ROS architecture and ROS2 architecture. Extracted from [45].

This approach naturally fits the purposes of a research tool, as it is simpler to monitor and analyse 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.

It was undeniable that the framework had architectural limitations that could not be fixed using the same design approach [45, 25]. ROS2 proposed a complete refactoring of ROS's former architectural design (right-hand side of Figure 12), that makes use of an external communication middleware that can support the production needs of the outgrowing robotic systems [38, 13].

3.1.2 Data Distribution Service

DDS[2] is an **Object Management Group (OMG)** middleware standard, that eases the complexity behind creating and maintaining communication architectures. It handles relevant aspects such as network configuration, communication establishment, data sharing, and low-level details. As a result, system developers can mainly focus on their application purposes rather than concerning about information moving across levels.

DDS architecture uses the **Data-Centric Publish Subscribe (DCPS)** model as its communication model approach. DCPS is based on a publish-subscribe pattern in which the *data-centric* messaging technique is implemented. It conceptually creates a virtual *Global Data Space*, accessible by any DDS-based application, where data is properly delivered to the applications which request for it, saving bandwidth and processing power [51]. A *domain participant* enables an application to participate in the global data space, either as a *publisher* or as a *subscriber*, according to its role on data exchange [45, 1].

Briefly, DDS leverages the premise of a transport-independent virtualised *Data Bus* (Figure 13) to address the 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 reliable 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.

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

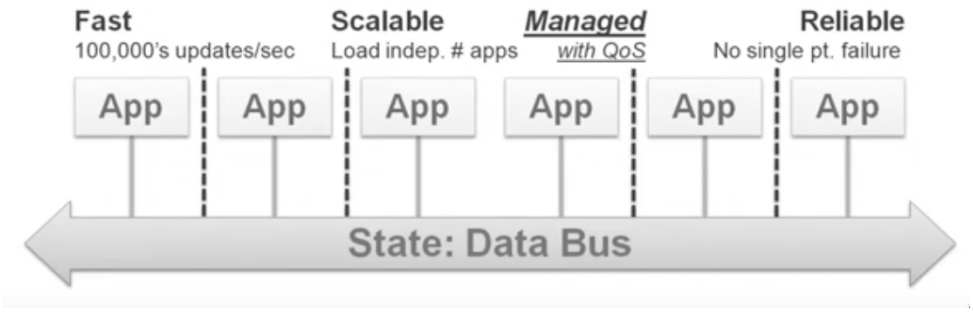


Figure 13: DDS communication in a nutshell. Extracted from [28].

3.1.3 ROS2-DDS Architecture

ROS2 middleware architectural approach (Figure 14) is built upon the DDS framework [45], leveraging DDS for its messaging architecture, where communication and transport configurations are handled. This enables real-time systems to be implemented reliably.

The middleware's top layer regards the **ROS Client Library (rcl)** [3], which has already been implemented in the former ROS architecture. This library exposes the concepts from ROS to the Application layer, as it provides APIs to ease software implementation by ROS developers [54]. The *rcl* hides these concepts by abstracting their specification, thereby reducing code duplication [60, 13].

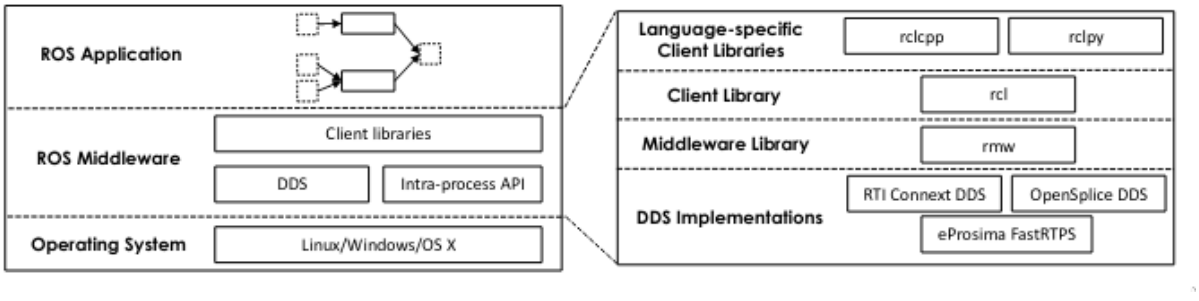


Figure 14: ROS2 framework architecture. Extracted from [13].

Multiple DDS implementations are supported by ROS2 to cover a wide range of applications. These differ in their API specification, which is strictly defined by its vendor. DDS defines fundamental procedures at a higher degree of abstraction, in which each implementation bases its API functionalities on.

ROS Middleware Interface (rmw) [13] abstracts these different implementations to *rcl* [13], allowing portability among DDS vendors. Consequently, it enables ROS developers to use several DDS implementations based on their application needs during runtime.

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

3.1.4 Computation Graph

From a logical perspective [13], ROS applications are composed of many software modules that operate as computation nodes, allowing their participation in the ROS global data space. The use of the publish-subscribe as communication paradigm, through message-passing patterns, confers additional complexity to the application architecture, where the latter can be naturally represented as a computation graph [20].

The application's computation graph presents itself as a graphical network, where runtime named entities have a unique role in data distribution.

Consider this specification of an application as a guiding example for the concepts that follow: a program rings an alarm continuously until a remote sends a message to make it stop. After receiving the message to stop, the program confirms that it turned off the alarm by sending an acknowledgement message to the remote.

Node Instances

Application development is done via package orchestration. Packages might comprise numerous nodes that can be perceived as processes that will likely perform computations over the network. Nodes can be connected within a single package or between multiple packages [20, 52].

Thus, the network is comprised of many nodes, running simultaneously and exchanging data between them, where each node addresses its corresponding purpose [54]. Fault tolerance features are guaranteed as nodes have their corresponding unique name, allowing communication in an unambiguous manner, which confers a suitable approach for developing a complex robotic system.

The use of callback functions associated with received messages provides great functionality when it comes to managing a node's behaviour in the communication process. Additionally, *timers* can also be used because they provide a useful way to manage these callbacks by time assignment.

In our example, there is a clear distinction between the remote and the program that runs the alarm. These are identified as the nodes of the application network.

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 characterises every data structure concerning information exchange between nodes. A message is defined by its data type, also known as its *interface*,

which can either be primitive (*integer*, *string*, *boolean*, among others), or defined by a complex data structure, where multiple data types are assigned to their corresponding fields [54, 52].

The ROS computation graph provides three different ways of establishing node communication, those being *Topics*, *Actions*, and *Services*. These communication mechanisms have different interfaces, specified in different folders with unique namespaces [54].

Topics are perhaps the most common method, naturally perceived as middle-communication buses, over which messages are passed through. Communication through topics is handled by the publishing-subscribing pattern. A node publishes the message to any number of topics, which are subscribed by nodes that want to get access to that message. Topics provide a multicast routing scheme, where published data is cast into the multiple nodes that are subscribed to the topic [13].

In our example, we must provide a method to capture the stopping and the acknowledgement message sending. So, the take here is to create two topics do deal with these type of messages. To create end-to-end topics, there must be at least one subscribing node and one publishing node. So, to send the stopping message, the remote creates a publish call to a topic *stop*, while the program creates a subscribe call to that same topic. So that, when the remote sends a message through *stop*, the program will receive the message. Analogously, we create another topic to treat the acknowledgement message; i.e, a *ack* topic is instantiated in a publish call in the program, and in a subscribe call in the remote.

In ROS, whenever a node creates a publisher with the intention of publishing messages through a specified topic, *roscore*[4], the application that implements the Master node is used to advertise the latter, enabling message passing to the corresponding topic subscribers. However, in ROS2 the discovery mechanism is decentralised, due to the DDS distributed architecture, so *roscore* is no longer used [54].

Even though topics are the most conventional way of communication owing to their multicast scheme, subscribers cannot be identified by publishers, so logging and synchronisation becomes rather difficult [52]. The use of services enables a client node, that can also be seen as a topic subscriber, to request data from a server, that likewise a topic publisher, provides data through a service. This is a bidirectional synchronous form of communication based on the request-response pattern.

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

[4] http://wiki.ros.org/roscore

Figure 15: Network topology of our guiding example.

Launch-Files

A conventional way of deploying a ROS application is through the use of *launch-files*, which are XML files that allow the configuration of entire robotic applications, where network nodes can be individually pre-configured. ROS makes use of the *roslaunch*[5] utility to automatically initialise the whole network, launching each node in the launch-file [52]. This provides a simpler method for starting the system nodes.

Nodes are identified using the *node* tag. The latter is linked to a specific package (*pkg*) and executable (*exec*). The *name* field enables the specification of distinct namespaces for each executable. Distinctive namespaces allow the system to start the nodes without any name nor topic name conflicts.

Additional node configurations, such as name remapping and parameter adjustments, can be specified using the *args* tag, which offers great functionality and flexibility to the launching process. Remapping represents a powerful ROS feature that allows multiple configurations under the same network launch-file. Node remapping enables the reuse of the same node executable in other areas of the system, whereas topic remapping allows for behavioural mapping across distinct topics [54].

In our example, after coding our nodes and their calls of publish and subscribe to the topics *stop* and *ack*, we can launch the application using a launch-file as it follows:

```
<launch>
    <node pkg="my_pkg" exec="my_program" name="program"/>
    <node pkg="my_pkg" exec="my_remote" name="remote"/>
</launch>
```

Listing 3.1: Launch-file of our guiding example.

The launching of the application means that each node of the application is actively running, and the topics are now active connections. This creates a network topology that can be represented as in Figure 15.

[5] http://wiki.ros.org/roslaunch

Parameters

Another relevant concept behind ROS is the existence of node parameters, that can be used, for instance, for the configuration of the network nodes. In the former version of ROS, the node parameters are controlled by a global *Parameter Server*, managed by the corresponding ROS Master [52]. However, in ROS2 each node declares and manages its own parameters, by making use of a callback function [54].

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 performance of the network [54].

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. It eliminates message forwarding inefficiencies while preserving the modular nature of ROS nodes [54].

Inspired by the former idea of nodelets, ROS2 introduces *components* as a software code compiled into shared libraries, that can be loaded into a *component container* process at runtime in the network, ensuring node composition [54].

3.2 Network Security

The deployment of real-time systems implies a special care about safety and security [45], resulting from demanding time-critical scenarios. Robotic systems fall under the umbrella of this broad system definition as they feature unique cyber vulnerabilities related to their integration over highly networked environments, which confers great importance on exposing critical time-reliant scenarios [46, 24].

3.2.1 Security Problems in ROS

The network security evaluation in a system is done by applying several analysis methods. Generally, these do not cover every security aspect, as new vulnerabilities arise from technological evolution [37]. The application of security countermeasures techniques to configure a system's network is a critical step when trying to achieve security. Network security entails the pre-exploration of the system's network through practical networking security procedures, such as intrusion detection and traffic analysis [44].

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

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

Moreover, there were also worries regarding the way ROS handled node communication. Network security may be jeopardised, as a result of the publish-subscribe pattern transparency, where node-to-node communications are done in plain text, exposing data content to unauthorised usage [38, 67]. Additionally, the former API did not confer any message integrity technique, making applications vulnerable to packet sniffing and man-in-the-middle attacks [67].

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

3.2.2 DDS-Security Specification

OMG accounts for security integration in DDS by supplying an in-depth specification and consequently adding features to the previously developed DDS standard. The DDS-Security is a specification that serves as a security extension to the DDS protocol, defined by a set of plugins (Authentication, Access Control, Cryptographic, Logging, Data Tagging), combined in a **Service Plugin Interface (SPI)** architecture (Figure 16) [5, 27]. This specification defines a *Security Model* to be enforced by compliant DDS implementations via invoking the SPIs [50].

Authentication

To impose a secure environment over the DDS Global Data Space, data integrity cannot be prone to unauthorised usage. Therefore, data exchange requires verification procedures to accurately identify the authenticity of each DDS domain participant.

The Authentication plugin is the most important plugin of the entire SPI architecture, as it provides means to validate the identity of each application, later regarded as domain participants [50, 27].

Each participant must be authenticated prior to entering the data space [69]. To implement such secure environment, the plugin relies on a **Public Key Infrastructure (PKI)**. This is in charge of issuing asymmetric keys to each participant, a *public* and *private* key respectively, that are used for both authentication and encryption procedures [26, 27].

The communication establishment over different participants must be preceded by a mutual handshake, where keys and certificates are exchanged to guarantee their authenticity [69, 38]. Additionally, the DDS permissions of a domain participant are also considered in this handshake. The control over

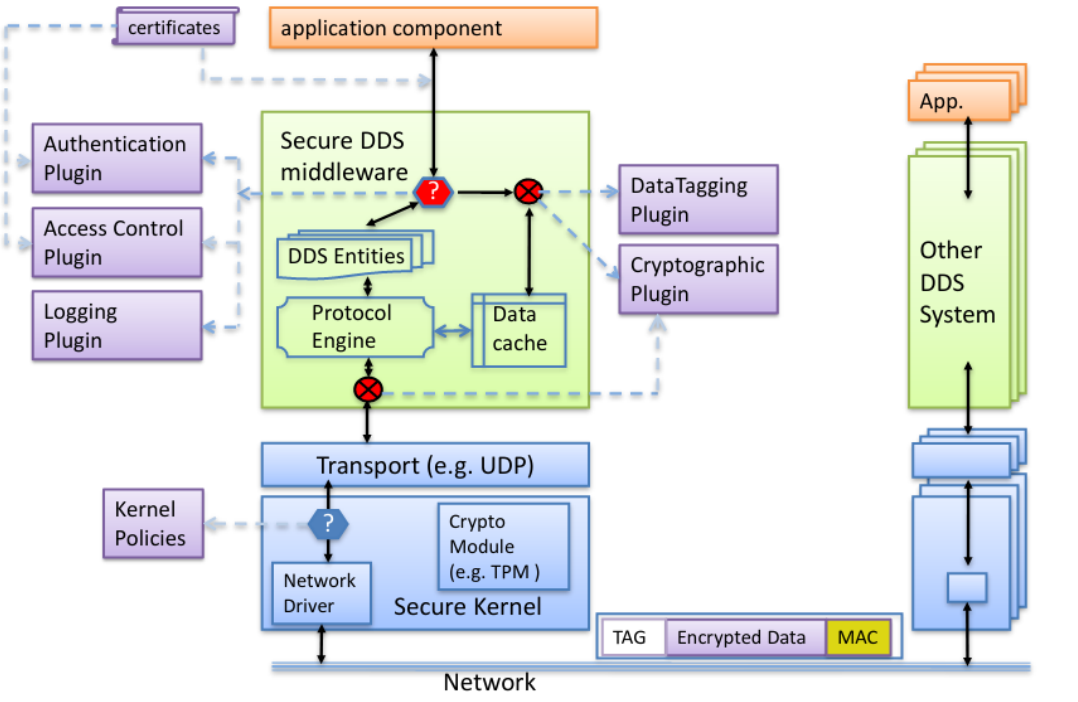


Figure 16: DDS-Security Architecture. Extracted from [50].

permission distribution is handled by the Access Control plugin.

Access Control

As aforementioned, the DDS specification implements policy control over the DDS domain through the Access Control plugin, where authenticated parties operations are controlled by policy restrictions [50, 69]. Domains within the Global Data Space are controlled with a set of DDS-related capabilities, which are either assigned or restricted to authenticated participants [27].

Authenticated participants must be granted access to certain domains, where their roles on data transportation must be specified by access permissions. If a participant is perceived as a domain data publisher, the domain restrictions must provide publishing privileges to its data topic [69].

Following the authentication procedure, domain authorisation is also implemented using the PKI [27], by embedding policy definitions through certificate extensions, signed by a trusted CA. This provides protection against the elevation of privilege attacks. If the policy integrity is jeopardised, the handshake establishment between authenticated parties will fail [67].

Communication and Encryption

The DDS-Security specification ensures encryption and authentication using *OpenSSL* [61]. Accordingly, the OpenSSL protocols **Secure Sockets Layer (SSL)** and **Transport Layer Security (TLS)** are used to ensure encryption over the network communication [67, 38]. The handshake is used to achieve mutual authentication within participants over the DDS domain [69].

The *Diffie-Hellman* key exchange protocol is used for the aforementioned handshake, allowing both participants to exchange data over a shared secret key [38]. It enables the exchange of both asymmetric keys, issued to each domain participant during the domain authentication procedure to perform sign (using the private key) and verify (using the public key) operations [38, 26].

After communication is established through this handshake process, the DDS-Security specification takes advantage of the *AES-GCM* encryption standard (the **Advanced Encryption Standard (AES)** with additional authenticated encryption functionality) to ensure data encryption over the channel [38, 61].

Every cryptography-related operation is handled by the SPI's Cryptographic plugin. Naturally, these cryptographic capabilities, linked to asymmetric key cryptography, are then used by both Authentication and Access Control plugins to ensure authenticity support over their corresponding operations [27, 26].

3.2.3 DDS-Security Integration in ROS2

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

Every DDS implementation supported by ROS2 makes use of the DDS-Security specification, enabling security over ROS's application environment. Even though ROS2 is deployed without security mechanisms by default [27], ROS2 provides a toolset, the **SROS2** toolset[6], that extends the set of command functionalities of ROS2 to make use of the DDS-Security functionality.

The control over these tools is done by the *rcl* client library, providing security to the Application layer, while DDS is capable of providing security to the communication architecture [38]. SROS2 configuration is done by defining a set of security files for each ROS2 participant with regard to how DDS handles certificate assignment [67, 27].

This toolset allows certificate generation and allocation through a keyserver directory, where security

[6] https://github.com/ros2/sros2

enclaves and their respective authentication files are stored inside subfolders [67, 27]. Moreover, SROS2 enables the keyserver configuration in a flexible way, allowing developers to restrict certificate and CA attributes to an arbitrary number of nodes, defined over the same namespace [67].

The variety of capabilities in SROS2 toolset attempts to aid with security configurations 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 can be a demanding task [27, 69].

Security Enclaves

As aforementioned, ROS2 relies on how DDS handles security over its *Domain Participants*. DDS imposes the authentication of each participant prior to joining its Global Data Space, using a **Certificate Authority (CA)** and a PKI to manage certificates [69, 67].

The authentication process within a ROS2 network relies on the notion of a network *enclave*, where authentication and control artefacts are stored to properly achieve security over the network data space [65]. Conceptually, an enclave is a secure region in the application address space that maintains confidentiality and integrity, while computations are being carried out on data.

Previously, a node was perceived as a separated DDS participant. However, with node composition, making each node a participant is not viable, because of the non-negligible overhead of combining different participants in a single process. Additionally, each participant's memory space becomes rather difficult to handle [65, 66].

Concerning the enclave authentication procedure, its security artefacts must be addressable by a DDS participant, where the latter is associated to a node sharing context, instead of being perceived as a single node [65]. Thus, when it comes to applying different policies over different nodes, that are matched in the same node context, a set of node profiles can be specified under the enclave notation.

Access Control within Enclaves The SROS2 toolset offers a *XML schema* [66] where security policies bind profiles to access permissions for network objects, granting privileges back to a certain profile. *Profiles* are implemented under the *enclave* declaration, to duly support the node composition into a single process, enabling the possibility of combining multiple profiles, respectively addressing its corresponding node.

Objects are classified over a subsystem type, structurally characterised by permissions tags. Then object privileges are controlled with access values, either *allow* or *deny*, attributed to their corresponding permission tags [66, 68]. Briefly, each node profile is assigned to policies that concern some **Object**

Identifier (OID). Each OID maps to a specific action that is identified with an access value, allow or deny respectively [67].

The policy design approach is *Mandatory Access Control*, that denies any privilege by default. The only way of allowing access to any object is by explicitly specifying the subject's privilege of access [66, 68, 67]. Naturally, this policy approach confers improved security, since it denies any attempt of privilege gaining attack by ROS-based packages from untrusted sources [67]. An example of a ROS policy file can be seen in Figure 17.

3.3 Related work

As reliance on robotic systems grows due to their expanding application across a wide range of domains, these systems are applied in critical scenarios, namely involving human interaction [26]. Obviously, safety should be highly considered when using robotic systems that might harm humans. Additionally, analysis regarding the system's quality assurance is increasingly becoming a focus of attention. Consequently, it is critical to employ techniques that promote the system quality without sacrificing flexibility.

Even though this dissertation is devoted to ROS2 security, there are many similarities to other robotic systems in terms of quality and security assurance. As a result, a broader range of studies are reviewed in this section to contextualise some relevant aspects that could also be applied to the ROS2 security domain.

The subsection 3.3.1 provides a comprehensive overview of the works that aim to prevent security issues in the ROS architecture. It is then followed by the subsection 3.3.2 that concerns previous work addressing property verification and model checking-techniques for robotics.

3.3.1 Security in ROS

Robotic systems were initially conceived as augmented computers with no explicit boundaries or limitations. As a result of the requirement to provide practical systems as fast as possible, security matters were disregarded [68]. Network security entails a cautious analysis of a system's network using realistic networking security procedures [44]. Concerning ROS and its role as a robotic application middleware, numerous researchers have examined the use of such procedures to perform a thorough security analysis of its architecture.

Some researchers used exploiting techniques [25, 21] to show the relevance of security in ROS, in particular to show how the *XML-RPC* embedded API can be breached and subsequently exploited according

```
<?xml version="1.0" encoding="UTF-8"?>
<policy version="0.2.0" xmlns:xi="http://www.w3.org/2001/XInclude">
  <enclaves>
    <enclave path="/private">
      <profiles>
        <profile ns="/" node="node_1">
          <topics publish="ALLOW" >
            <topic>topic_1</topic>
          </topics>
          <topics subscribe="ALLOW" >
            <topic>topic_2</topic>
          </topics>
        </profile>
        <profile ns="/" node="node_2">
          <topics publish="ALLOW" >
            <topic>topic_2</topic>
          </topics>
          <topics subscribe="ALLOW" >
            <topic>topic_3</topic>
          </topics>
        </profile>
        <profile ns="/" node="node_3">
          <topics publish="ALLOW" >
            <topic>topic_3</topic>
          </topics>
          <topics subscribe="ALLOW" >
            <topic>topic_1</topic>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

Figure 17: Example of a SROS2 policy file. Privileges are distributed across the enclave and their profiles.

to the participants roles within the network. Besides raising awareness of the importance of security in ROS, these works alerted developers for the need to conduct penetration testing on their applications.

Following the challenges that arose as a result of executing exploitation techniques on the ROS framework, numerous solutions [46, 23, 8] have been proposed in response to the need to offer security assurance for robotic applications.

In [67], White et al. address the security deployment over ROS in a more concise way, by proposing the **Secure Robot Operating System (SROS)** as an enhancement to the former API, that includes mechanisms such as authentication, access control and cryptography measures. SROS seeks to provide an adequate security architecture while minimising disturbances such as computational cost and API breaking. SROS allows developers to customise it to their own internal certificate formats. Moreover, White et al. expect security logging and access control to evolve through well-defined standards, enabling more robust auditing and policy generating tools. White and colleagues continued to provide security mechanisms to the framework in subsequent studies [11, 68, 69]. In [11], Caiazza, White, and Cortesi present several solutions for the lack of security measures that autonomous devices, often related to the robotics world, tend to face. The effort then moved on to evaluate the ROS framework in order to provide a high-quality solution. It follows a pipeline of security measurements, where logging, access control, and authentication certificates are discussed over several proposals.

In [68], White, Christensen, Caiazza, and Cortesi address robotics security through a framework that focuses on handling access control policies for robotic software, with the intention of adding functionality to SROS [67]. The latter offers an interesting perspective on leveraging access control through a well-typed markup language schema, the *ComArmor* configuration language, where privileges of objects are explicitly defined over policies. Profiles are then used to arrange these policies in a hierarchical manner, binding namespaces to object privileges using attachment expressions with predefined permissions denoted as rules.

Following the deployment of DDS in ROS2, the majority of the security problems were alleviated. As a result, the literature on network security enhancements provided by ROS2 is largely concerned with providing an overview of the trade-off between performance and security. Furthermore, various studies evaluate ROS2 performance in terms of response time under safety-reliant circumstances [45, 13, 38], in which DDS capabilities are highly emphasised to properly evaluate ROS2.

A pertinent research that examines the trade-off between performance and security is presented in [26]. Here, DiLuoffo et al. conduct a thorough examination of ROS2 and outline possible hazards for this new robotic system middleware. The DDS-Security specification is thoroughly examined in terms of

performance versus security models and how security is integrated with ROS2, as well as how the addition of security affects system design. This experiment illustrates that implementing security measures on protected data in motion results in a significant performance loss. Notably, they conclude with some recommendations towards security areas not duly covered by the DDS specification, such as attacks stemming from spy processes. They highlighted the preservation of the cognitive layer because of its importance in the robot itself, since all types of sensors that involve data collecting are part of that layer. Here, an advanced set of threats is described using Machine Learning, which DDS is not capable of mitigating.

These works present several considerations about how important is to address security in robotic systems, due to the variety of attacks that might compromise their integrity. They follow the path of addressing vulnerabilities through applying security methods based on tools and protocols. Studies regarding the application of static quality analysis and formal property verification in ROS are quite limited. Despite this, the next section reviews several works that advocate using formal techniques to assess robotic systems.

3.3.2 Verification of Robotic Systems

Property verification related to the security configuration in ROS2 is the focus of this dissertation; however, researches and tools for the analysis of ROS systems are scarce, there are some works that might be useful within the context of this dissertation.

Cortesi et al. [19] give a proper explanation on how important it is to perform formal analysis to ensure the reliability of the resulting software, while significantly minimising the testing effort. In this work, the authors intend to cover multiple static analysis techniques to properly contextualise robotic developers about which is the most appropriate analysis strategy to follow based on the kind of examined property and software system. After briefly introducing each technique, the authors present an overall evaluation of the various techniques, following several criteria: automation, precision, scalability, and soundness. Cortesi et al. conclude stating that performing analysis techniques is highly fundamental to develop robotics software. Additionally, the authors discuss the difficulties in guaranteeing an adequate automated analysis of complex properties. Finally, they emphasise the need for future research within this formal verification context because of its role in software development.

The **High-Assurance Robot Operating System (HAROS)** framework, initially proposed by Santos et al. in [56], has the overall goal of improving ROS software quality. HAROS makes use of static analysis to evaluate ROS software. On its own, it is able to provide feedback about code quality by presenting some predefined code metrics. Santos, Cunha, and Macedo further extended HAROS capabilities through the

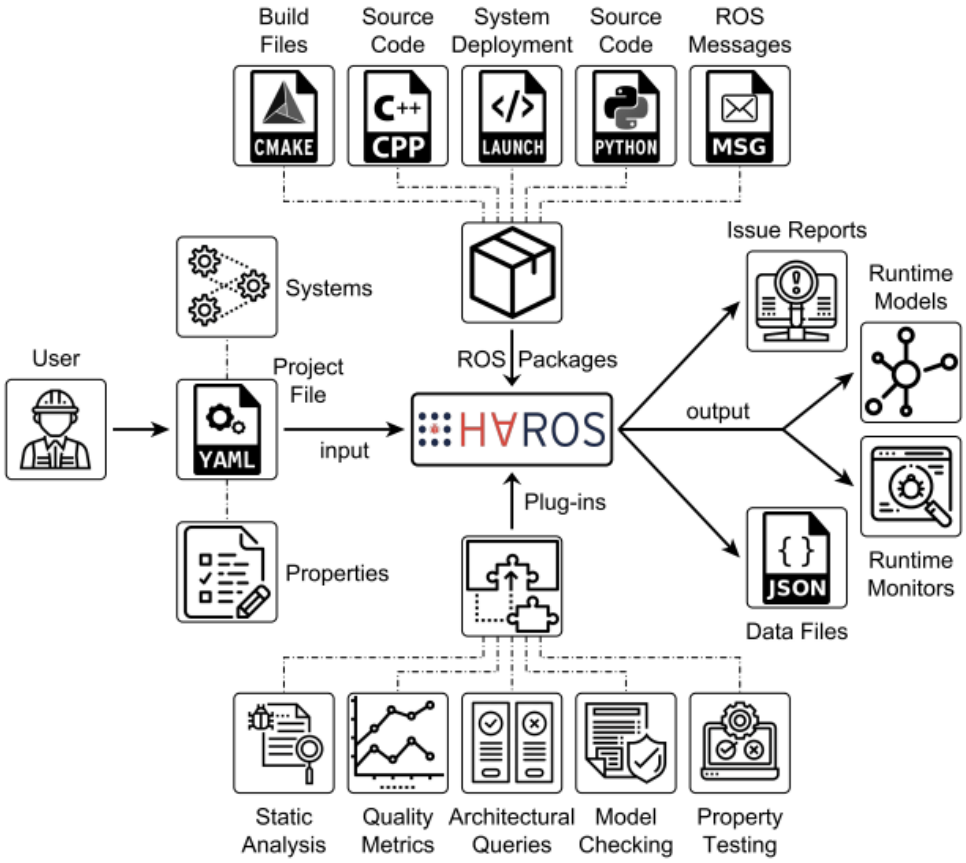


Figure 18: Workflow of the HAROS framework. Extracted from [59].

implementation of several plugins [59, 57, 58] that enable other sort of analysis. The authors explain why covering a wide range of analysis techniques is essential, stating that adopting one technique in isolation might not be ideal nor sufficient.

To put it briefly, in HAROS (Figure 18), the user provides a YAML-based project file, which specifies the configuration launch-files that represent concrete applications to be analysed. Additionally, behavioural properties can be specified to be evaluated during the analysis phase. HAROS then locates ROS-based packages, where the source code is extracted and subsequently parsed, yielding a complete graph-based model. These models not only provide developers with visual feedback on the application architecture but also enable automatic analysis. The analysing process is carried out by any installed analysis plugins.

A general overview of property verification through model-checking was already presented in Section 2.1. Within the ROS context, certain techniques have been proposed that primarily focus on modelling ROS node communication with the goal of supporting model-checking of desired properties.

Within this context, Halder et al. [33] provide a model-based approach to reason about the communication between nodes in ROS-based systems, emphasising the specification and verification of real time

properties. Here, the authors model ROS applications by abstracting them as timed automata. Property verification is then ensured by model-checking through the *UPPAAL*[7] model-checker. The authors applied this technique to the popular *Kobuki* robot[8]. This allowed the authors to reason about context specific properties of the latter, aside from the safety properties.

A work that is very relevant to this dissertation is presented in [12]. Here, Carvalho et al. present a technique to automatically verify system-wide safety properties of ROS applications at static time. The proposed technique abstracts each application behaviour through a concrete model, to then perform model-checking of system-wide specifications using *Electrum*[9] as model-checker.

To integrate the analysis of ROS-based applications, the authors make use of the already mentioned HAROS plugin-based framework. As already explained, HAROS allows the automatic extraction of the system architectural model from the source code and launch-file configurations. Then, using a domain-specific property-based language allows the specification of the desired properties and assumptions about the nodes' behaviour. The proposed technique relies on Electrum, the former version of the current Alloy Analyzer, to model and reason about the application behaviour through properties expressed in FOTL. This technique is offered as a HAROS plugin that automatically converts HAROS artefacts into Electrum. Additionally, the plugin reports model-checking counterexamples in a more user-friendly manner via the HAROS quality reporting interface.

For the evaluation, the authors applied their approach to a real ROS-based robot. The results show that the specification language is sufficiently expressive to address the relevant properties of the system. Then, they address the scalability by evaluating the technique performance. As future considerations, the authors propose techniques to help discard false-positive counterexamples that can emerge owing to the use of an abstract model, namely, because no particular scheduling is imposed on the message-passing process.

As ROS2 relies on 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 communication protocols.

In [1], Alaerjan et al. propose 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 [41], DDS in ROS2 is formalised as timed automata, followed by model verification.

7 https://uppaal.org/

8 http://kobuki.yujinrobot.com/

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

In [69], White et al. continue to address security through an alternative approach. Here, the authors make use of formal verification and model-checking to reason about reachability of information flow related to an attack model that studies the network through targeted attacks. This study reviews the DDS security protocol, as well as its model, and concludes that it lacks support for protection against security threats.

Within this scenario, White et al. intend to explore the data flow semantics to address the DDS networks vulnerability. After conceiving the threat model, some attack assumptions are explicitly defined in the attack model. Unlike previous network reconnaissance methods and considering the assumptions discussed in the attack model, this approach allows an attacker to build the topology through a graph based database by passively sniffing packets inside the network. This database is then made available to a SAT solver for computing queries. The *Imandra*[10] framework is used as SAT solver after replicating the DDS security protocol functionalities (especially access control) as functional models to accurately represent the default plugin logic. The attack model assumptions are specified as SAT-based queries, allowing the replicated models to reason about them. Moreover, using formal verification and model-checking, vulnerability detection can be efficiently performed over the inquired graph, instead of engaging with the targeted system.

Another relevant work regarding analysis using Alloy specification language is presented in [43], where a safety-critical scenario is proposed in the domain of surgical robots. This formal technique was applied to a surgical robotic arm, considering possible violations of important safety properties. Although this study presents favourable results, it focused on a particular case-study, lacking the generality to be applied to the vast majority of robotic systems.

[10] https://www.imandra.ai/

Chapter 4

Verification of Security Properties

Security is a very broad subject with many implications in the deployment of robotic system applications. Despite the deployment of techniques used to deny exploiting attacks, such as control of access and supervised authorisation, security gaps in software are a unsustainable burden. However, techniques that resort to formal methods and property verification present a great approach for ensuring security in software development.

Conventional model-checking does not directly address security properties because most security policies cannot be expressed as single trace properties. They require relating multiple traces rather than a single trace, representing properties of sets of execution traces. These are called Hyperproperties, and this chapter presents relevant background to contextualise the reader about why they are crucial to the verification of security.

This chapter starts by contextualising the problem by presenting an academic ROS2 application, that has no clear-sight security flaws. The take here is to show the importance of hyperproperty verification in security assurance. It focuses on a particular security property to show the viability of the proposed technique, namely **Observational Determinism (OD)**.

It follows with the definition of Hyperproperty, and the relevance of OD in distributed systems, and how self-composition can be used for its verification. Lastly, it is described how Alloy can implement this technique in the case-study application.

This work focuses only on topics since the publish-subscribe paradigm is the most relevant means of communication in ROS2. Even though services usually present great value in ROS2 applications, they work synchronously between both ends, which is not convenient for this type of flow analysis.

4.1 Problem Contextualisation

Consider the ROS2 architecture in Figure 19, which is referred to as *TurtleSim*. A secure or private enclave is defined as one in which all the trusted nodes operate. Nodes outside this enclave are public and untrusted (*Random* for instance); therefore, it is important to discern between events that can be trusted and those that cannot. The goal is to show that the secure enclave appears to behave in a deterministic way for those public nodes, thus ensuring the confidentiality of its private information.

Random tries to stir up the turtle movement by sending movement messages to the enclave repeatedly. The turtle position is altered each time a movement command is issued to it, and it is enclosed in *Turtlesim*. The latter is labelled as a trusted node, which means that the turtle position is set as private. Only trusted nodes can directly communicate with the turtle interface and inquire about its position.

The turtle is supposed to operate inside a safe region. When the turtle hits the border, the node *Safety* is responsible for directing the turtle away from it, by publishing commands of movement.

Multiplexer is responsible for handling movement commands issued from either *Random* or *Safety*. *Multiplexer* chooses which command to forward to *Turtlesim*, triggering the turtle's movement. *Safety* has priority over *Random*, meaning that when *Safety* publishes movement data through its corresponding topic, *Multiplexer* must instantly process the received data.

Additionally, consider the following public output of the secure enclave. If some of the received commands handled by *Multiplexer* have *high* values of movement, *Multiplexer* will trigger an alarm light's switch on. Triggering is achieved by sending of a message to the outside through the *alarm* topic.

The idea is to show that *Random* cannot possibly know anything about the private state of the system, namely the current position of the turtle. So, the question is: what can *Random* use to infer something about the position of the turtle? Since *Random* can only see the alarm light, because it denotes the only observable output aside from its own output, the only way for private information to be leaked is through pattern matching between its own output and the *alarm* output (whether it turns on or off).

Therefore, to ensure that *Random* is unable to infer such information, it suffices to ensure that the system appears deterministic to *Random*; that is, it suffices to check the following property:

> The same state of the alarm light is observed whenever the same movement command is inputted to the secure enclave through topic *move*.

Obviously, if *Safety* issues *high* values of movement, the alarm will trigger. Consequently, if *Random* only sent *low* values of movement, it could easily be inferred that the turtle hit the border. Thus, to ensure security, *Safety* can only issue *low* values of movement.

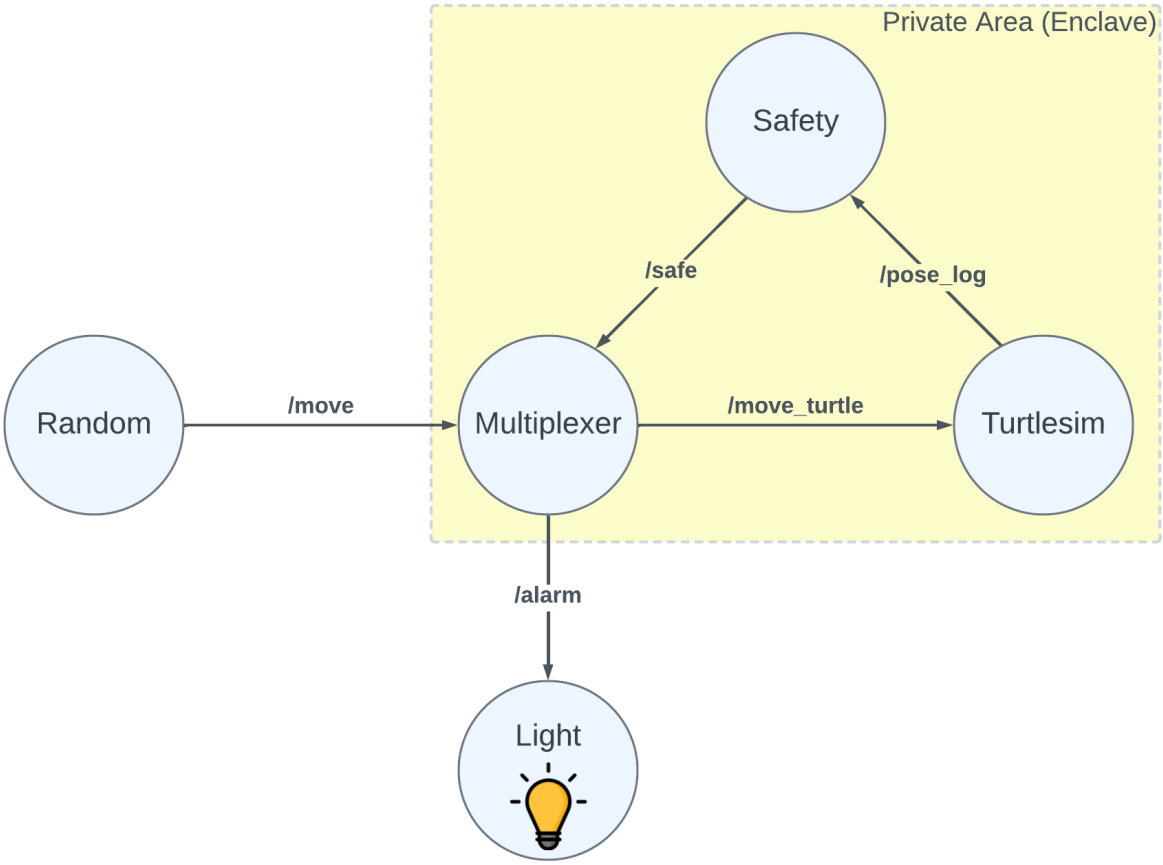


Figure 19: *TurtleSim* computation graph. Topics are represented as pointing arrows, from the publisher node to the subscriber node.

Surprisingly, as shown later, even with this restriction, it is still possible for *Random* to infer information about the turtle position, which means that the alarm behaviour does not entirely rely on the same sequence of events. In another words, some internal nondeterministic data handling must occur for this to happen.

The property cannot be expressed using the conventional linear temporal logic because it requires comparing different execution traces. Its expressiveness falls under the domain of *hyperproperties*, which are temporal properties whose validity can only be verified on sets of traces and not on a single trace.

The deterministic appearance to public entities is exclusively verified when there is a direct interaction with private parties. It is clear that if no public node issues inputs to the private enclave, they cannot infer any relation between their own outputs and the enclave's outputs. On the other hand, if the private enclave does not output any value to the outside, no public node, for instance *Random*, is able to exploit private information.

4.2 Hyperproperties

Likewise temporal logic properties, *hyperproperties* also uses some sort of temporal logic language to impose constraints on executions through predicates. However, a hyperproperty [17] is more difficult to interpret and express than a normal property, as it generalises the latter by stating a predicate on sets of executions, establishing logical relations between multiple executions of a system, whereas a property denotes a predicate over a single trace of execution [18, 40].

Hyperproperties have become relevant in the verification of security flow in a system [64] because temporal properties are not expressive enough to specify such policies. Model checking tools, such as Alloy, were developed to target systems where correctness properties (*safety* and *liveness*) related to the behavioural evolution of the latter were specified using classic temporal logic, such as **Linear Temporal Logic (LTL)**, so they cannot be used to directly verify hyperproperties [18].

Consider a temporal property P and a system S, comprised by multiple traces t_i. Briefly, S satisfies P *iff* every trace t_i respects P; so, a temporal property P can be seen as a set of traces. As traces are a formal representation of a sequence of states, which denote system behaviours, it is plausible to say that a temporal property is a predicate on behaviours [40].

Consider a terminating sequence of behaviours as a black-box, that takes a single input of x and outputs a terminating value of y, which is specified as the follow condition H:

> Every time x is inserted into the black-box, the output y is the same.

It is clear that normal temporal properties are not sufficiently expressive to specify H because a counterexample to this restriction cannot be a single trace. The correctness of the property depends on every pair (x, y) appearing in different traces of the system.

In order to show that H does not hold, at least *2* traces must be produced with the same input x, but the output y differs, as shown in Figure 20.

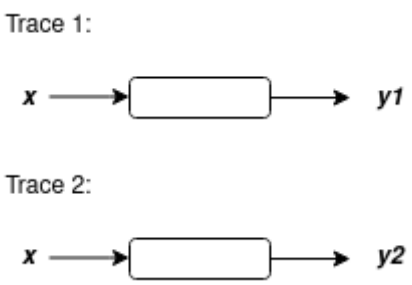


Figure 20: A counterexample to H.

Hyperproperties allow the specification of such conditions, since their validity is defined on a set of behaviours, rather than a single behaviour, consequently adding expressiveness to classic temporal logic [40, 17]. Property H is thus an example of a hyperproperty.

Property H can be seen as a security policy in which both inputs x and outputs y are labelled as untrusted or public events, and the black-box is actually a secure program p_s. H can now be restated as follows:

> Every time that a untrusted input x is consumed by the secure program p_s, the public output y must be the same.

Basically, the program p_s must appear as a deterministic transformation from untrusted x inputs to public outputs of y, meaning that the former has no direct impact on the outcome of p_s. Therefore, it is not possible to infer anything about the internal state of p_s.

4.2.1 HyperSafety

Likewise normal properties, hyperproperties can fall under the categories of either safety or liveness. However, it is reasonable to claim that safety hyperproperties have the most interest, as the bad thing that is being verified can be finitely observable. Thus, its occurrence might indicate that a specific security policy failed to hold. In terms of security, it is also more interesting to check that some hypothetical behaviour cannot occur (that being the "bad thing") than to specify a possible, indefinitely observable behaviour.

4.2.2 Observational Determinism

Hyperproperties treat data-passing between trusted and untrusted entities through security-related events, where the latter are classified as inputs or outputs. Additionally, events are labelled with a confidentiality level to information [17, 18], either public (also called low) or private (also called high).

Non Interference

A classic example of a hyperproperty is **Non Interference (NI)**. In a system with propagation of data from inputs to outputs, the goal is to prevent untrusted entities from inferring private information through environment interaction and output observation [17]. Public outputs can unintentionally lead to information leaks, if the private data is directly or indirectly involved in the computation of such outputs [55].

The NI policy, as defined by *Goguen* and *Meseguer* [31], is a safety hyperproperty that states the following:

> The public output observations are the same as they would be in the absence of private inputs.

NI enforces the following behaviour: given two execution traces $t1$ and $t2$, if $t1$ has no private inputs, and $t1$ and $t2$ have the same public inputs, then the public outputs must be the same.

However, the original formalisation of NI holds under the assumption that the system being checked is deterministic when it comes to input handling, and any concurrent system, such as the ones based in ROS, has many different nondeterministic interleavings.

Nondeterminism

Multiple formulations came after the original NI definition to consider nondeterminism within systems. The most popular policies are named **Generalised Non Interference (GNI)** and **OD**. The former, however, does not denote a safety behaviour and, consequently, cannot be verified efficiently (namely, using the self-composition technique presented in the next section). Thus, in this thesis, the only NI policy of interest is OD.

A nondeterministic system is said to satisfy OD if the outcome of its public events appears deterministic. Briefly, this policy states that after a public input is taken, further public observations have to be the same in every execution trace. Thus, if two traces t_1 and t_2 have the same public events, then further observations must remain indistinguishable for untrusted outsiders.

As defined by Zdancewic and Myers in [71] if those traces have the same initial observable state (or are public-state equivalent), then they must remain *public-stuttering* equivalent. Briefly, this means that the same public events occur in both traces at the same "time", except for the inevitable stuttering. This property can be formalised as follows [17].

$$\forall t_1 \cdot \forall t_2 \cdot t_1[0] =_{ps} t_2[0] \rightarrow t_1 \approx_{ps} t_2$$

where $t_1[0] =_{ps} t_2[0]$ means that the initial state of both traces are public-state equivalent and $t_1 \approx_{ps} t_2$ means that both traces are public-state equivalent up to stuttering. Later on, it is shown how this can be verified using self-composition and *lock-step synchronisation* of traces.

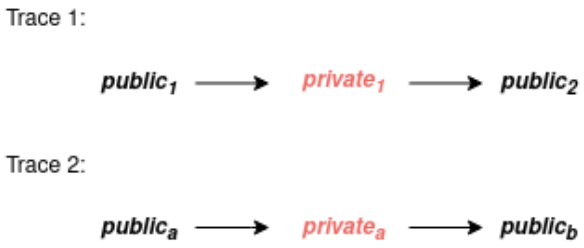


Figure 21: *Observable Determinism*: Trace 1 and Trace 2 executing in parallel.

As OD expresses a safety behaviour, its refutation can be directly achieved by providing a pair of traces where the non-intended bad thing happened during execution. In the example of Figure 21, OD holds if $public_1 = public_a$ and $public_2 = public_b$. Note that, if the latter is not satisfied, it means that the private events, $private_1$ or $private_a$ leaked some information to those public observations.

4.3 Self-Composition

Verification of properties in a system is a well-studied subject and reasonably simple to comprehend. Both properties and systems represent a set of traces. Recall the explanation given early on in this section: a system S satisfies a property P, iff every trace of S is an element of P, as P is a set of traces.

$$S \models P \quad \textit{iff} \quad \forall t \in S \cdot t \in P$$

On the other hand, a hyperproperty is a predicate on properties, which means that it specifies exactly the allowed set of traces. Thus, it can be abstractly represented by a set of sets of traces, of which a system S might be a part of. So,

$$S \models H \quad \textit{iff} \quad S \in H$$

This idea is rather simple to grasp in theory; however, its application in model-checking tools is not that straightforward, due to the nature of these tools. They were developed to be able to design systems and specify behaviour upon them through predicates on executions. To use model-checking, a mapping must be conveyed between hyperproperties and properties, stating how a hyperproperty H can be verified by an equivalent property P.

Consequently, a system S must be mapped into another system $\omega(S)$, in which the validity of P entails the validity of H in S. Bare in mind that H is a predicate on a set of traces of S. Naturally P, as

a property, is a predicate on a single execution of S, so $\omega(S)$ must be composed by multiple individual copies of S that execute simultaneously, in the so-called lock-step, for P to be a representation of H in such copies.

$$\omega(S) = S \times S \times S \ldots \times S$$

This concept of replicating a system S into multiple copies is known as *Self-Composition* [4]. It can be used to reduce the verification of some hyperproperties to the verification of normal properties.

Recall the example of OD presented in the last section.

> Every time that a untrusted input x is consumed by the secure program p_s, the public output y must be the same.

The representation of this OD hyperproperty through a property P is as follows:

> Any two terminating behaviours satisfying P that have equal initial values of x have equal terminal values of y.

Briefly, this means that if two different execution traces t_1 and t_2 of S begin with equal public events, involving the input of x, then they always output the same values of y. For that matter, in order to apply self-composition to the latter, only a pair of execution traces is needed to run simultaneously, so $\omega(S) = S \times S$.

4.3.1 Lock-Step Synchronisation

Replicating a system S into several individual executions implies that synchronisation between them must be duly assured. These executions are interpreted as traces of events that logically link a sequence of states [40]. As formal frameworks, including Alloy, abstract the progress of time as execution steps, public events must be synchronised so that stuttering is properly accounted [18, 40].

As mentioned, security-related events are either public (untrusted to the whole security environment) or private. Thus, the conventional take is to consider that executions running in lock-step are public-event synchronised, disregarding private and stuttering events [18]. In brief, n executions of a system are correctly synchronised if the same public event occurs at the same trace position (Figure 22).

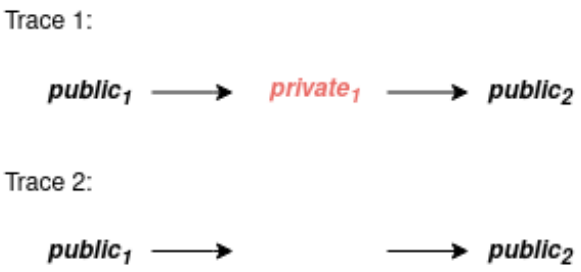


Figure 22: Lock-step synchronisation of two traces.

Most formalisms consider traces *Trace 1* and *Trace 2* to be different executions because of the interleaving between private and public events in the former. However, when considering OD, it is reasonable to consider them as a correct pair of traces, since the public observable state is the same, and hence public entities cannot really infer any concrete information about the private state, as *Lamport* states in [40].

4.4 Hyperproperties in Alloy

Alloy and other standard model-checkers were not designed to perform verification of hyperproperties; therefore, the reader must understand the limitations of adopting Alloy as the model-checker of choice in such analysis. Modelling hyperproperties in Alloy requires the use of self-composition, which results in a replication of the model state.

Replication has to be carefully handled. Replicating the whole model is not ideal, as this would duplicate every signature scope, which can substantially decrease performance. The best approach is to identify which model parts affect on how the analysis trace evolves; meaning, which signatures and fields are mutable (or time-variable). Only these need to be replicated in self-composition, leaving static signatures and fields aside.

Time-variable structures are those who have a direct impact on data-passing in topics and communication between nodes. Although topics are static, the messages they carry are not, as they are created in publish calls and discarded when broadcast. Application variables, such as the turtle position, are also relevant to how the trace evolves, so they should also be replicated by self-composition.

In the remainder of this section, the reader is instructed on how to approach modelling when using Alloy towards the verification of OD. We briefly explain on how the *TurtleSim* Alloy model can account for public-event (or lock-step) synchronisation, which is a relevant step towards the verification of OD.

4.4.1 Hyperproperties in TurtleSim

The reader must now understand that the property in which the system must appear deterministic to *Random*, is indeed categorised as the hyperproperty of *OD*. Remember that this requires equality on the relation between the inputs of the *move* topic and the outputs of the *alarm* topic.

> The same state of the alarm light is observed whenever the same movement command is inputted to the secure enclave through topic *move*.

The latter can easily be moulded into a hyperproperty that implicitly makes use of self-composition. Given any two replicas of execution traces $T1$ and $T2$,

> If the same movement command is issued by *Random* at the same time in both traces $T1$ and $T2$, then the output state of the alarm light will always be the same in both traces.

The avoided behaviour states that if those traces differ in the output of the alarm light, then internal computation within the enclave causes the difference between outputs. Roughly this means that, the final state of the trace is not an apparently deterministic function of the public inputs outside the enclave; thus, the latter can be exploited to leak information.

Public-State Equivalence

The verification of OD requires a fair environment in which trace replicas must have the same initial observable state. This is the same as being public-state equivalent. For now, consider that a public state is identified by variables that confine public information about the system. Particularly, in this example, apart from the turtle position, which is private, no other variable is outlined.

Public-Event Synchronisation

The trace replicas must continue to be indistinguishable for public nodes, which means, that they must remain equivalent in their observable state. This is referred as being public-event synchronised. Topics are the core of communication; therefore, an event is specified upon topics, and consequently, public events are specified upon public topics.

Public topics are topics in which one of the ends is public: in this example, those topics are *move*, through which *Random* issues velocity messages, and *alarm*, where *Multiplexer* outputs the state of the alarm. One differs from the other, though. The former is an input to the secure enclave, while the latter is

an output. OD assumes that the same message is issued in *move*; however, in *alarm*, instead of assuming this equality on messages, it aims to check it.

So, when modelling public-event synchronisation in the *Turtlesim* example, the following conditions are assumed:

- The same message is published through *move* in each trace replica, synchronously.
- A message is published in each trace replica through *alarm*, synchronously.

Now that both conditions to verify OD in *Turtlesim* have been presented, the following section shows how trace replicas are represented and how self-composition is used in Alloy.

Self-Composition

The best approach for self-composition is to discern relations which are mutable from those which are not. Remember that mutable fields are the topics and application variables. Topics store messages in a buffer. Application variables, namely the turtle position, are also set as mutable, so they can have an impact on how a trace behaves. For instance, the trace state when the turtle hits the border is different from the trace state where the value of the turtle position is in the centre.

To simplify self-composition, we declare an Alloy signature that explicitly models execution traces: the *Trace* signature. The latter captures mutable signatures and relations, and declares them as its own signature fields. Remember that to verify OD, only two trace replicas are necessary. Thus, two singleton extensions, identified as $T1$ and $T2$ respectively, are instantiated. This approach makes the specification of the model less verbose than creating two copies of each variable field and signature. Listing 4.1 shows how Alloy captures this idea, where the field *inbox* identifies the relation between topics and their message buffer, whereas the *position* identifies the turtle position.

```
abstract sig Trace {
  var position: one Pos,
  var inbox:    Topic -> seq Message
}
one sig T1, T2 extends Trace {}
```

Listing 4.1: *Self-Composition* captured by the *Trace* signature.

From this point forward, the system evolves through these trace replicas. So, a system behaviour actually corresponds to a self-composed system execution behaviour. In Alloy, a system specification is composed by atomic events, each specified in a predicate; however, a trace t is now given as an

argument, and every access to a variable relation is projected to that t. In the next chapter, we provide in-depth information on how this is dealt with.

Verification of Observational Determinism

Remember that the property intends to verify that the same state of the alarm light is observed, which means that it expects that the same message is published synchronously through *alarm* in each trace replica.

Publishing through public topics, such as topic *alarm*, where the source node is private and the other end is public, are designated as public observations of the system, whose equality is what OD tries to verify.

The concrete verification assertion is presented later on in this dissertation, together with in-depth description on how these specifications can be obtained automatically.

Counterexample Unfortunately, the verification of OD in this case-study yields the counterexample in Figure 23. Both traces start with the input of the same velocity message m issued by *Random* (as expected because of public-event synchronisation). Here, command m has a *high* value of forward movement, denoted by **+2**.

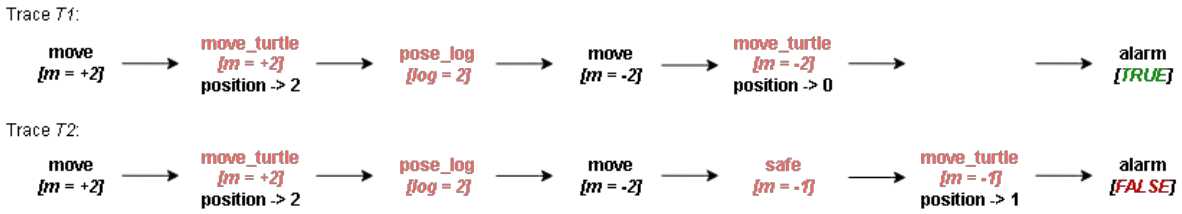


Figure 23: Counterexample with each trace $T1$ and $T2$ executing synchronously in lock-step. Red execution steps represent publishing through private topics, whereas the black ones represent publishing through public topics. Between square brackets the value of the sent message is presented.

Traces only appear to differ in the last three steps of execution. This difference, caused by interleaving and stuttering, causes the emergence of the counterexample. In trace $T1$, as *Multiplexer* issued a *high* value of velocity, a ringing *Alarm* message was issued. Whereas in trace $T2$, *Multiplexer* forwarded the message that arrived from *Safety*, which consequently sends a not-ringing *Alarm* message.

Solution It is clear that OD fails to hold due to difference between events within the enclave. Along with the non-equivalence between private events, interleavings and stuttering are the main causes of

nondeterminism. Following a thorough examination, a possible solution is to allow the system to still stutter but under a somehow synchronised environment within the enclave, where a specific processing order between nodes must be enforced:

> *Multiplexer* is only allowed to process its incoming messages, once both *Safety* and *Turtlesim* are clear of messages to be processed.

This strategy can be thought of as adaptive stuttering, because *Multiplexer* only stutters while it waits for the others to finish processing. In distributed systems, such as ROS's, the latter restriction is acceptable, as it represents the whole idea behind synchronisation between processes and the use of conditions to either lock or wait for others to finish.

Although the behaviour predicate of *Multiplexer* was omitted, this synchronisation can be ensured by adding the following guards to the respective event.

```
no t.inbox[topic_pose_log]
no t.inbox[topic_move_turtle]
```

Listing 4.2: Excerpt of the Alloy predicate of *Multiplexer*. This node can only execute when both `pose_log` and `move_turtle` boxes are empty.

Synchronisation in ROS Topics are not naturally synchronous. Typically, services are used to provide synchronisation between both ends, under a request-response architecture. However, according to the documentation [54], if the buffer size of a topic is set to *None*, communication between ends is handled synchronously. Thus, this is one way to ensure this synchronisation in ROS.

Another way to ensure this synchronisation is to run *Multiplexer* at a much lower frequency than the other nodes. Since the other nodes only react (direct or indirectly) to its messages, theoretically this would suffice to ensure that when *Multiplexer* runs, the other nodes already finished processing their messages.

Chapter 5

svROS - a ROS2 security verification tool

Verification of security in robotic systems is one of the most difficult tasks from the standpoint of software development, as it might lead to a variety of loose ends. However, it was shown how security hyperproperties, in particular OD, can be verified by resorting to the use of formal methods.

Using formal frameworks for verification, such as Alloy, requires a significant level of expertise, which a common ROS developer does not possess. In addition, no state-of-art tool contemplates techniques to formally verify security in ROS2, which naturally motivates the study considered within the scope of this dissertation.

Therefore, a verification tool was developed, named **Security Verification in ROS (svROS)**, which focuses on abstracting formal verification approaches, to provide a less-formal, easier to use, solution to verify OD in ROS2 system applications. To check the correctness of a ROS application behaviour in respect to OD, it is necessary to specify how the system behaves atomically in each node. For this, the tool incorporates a specification language that is more user-friendly than Alloy and, it enables the specification of intra-node operations, in respect to the publish-subscribe paradigm.

Considering the proposed objectives, the tool supports the following capabilities:

- Source code fetching from ROS2 application packages.
- Reverse engineering methods to infer an architecture topology from the extracted code.
- Generation of configuration file templates, to allow a ROS developer to easily configure its application network.
- Methods to translate the system configuration into a model in Alloy, to later perform the verification of OD.
- A domain specific language to specify the intra-node behaviour of a ROS application, and methods to translate such specifications into Alloy.

5.1 Security Verification in ROS

svROS verification tool aims towards optimisation and automation when it comes to the overall extraction and analysis of robotic systems developed in ROS2. The technique firstly focuses on migrating existing static analysis methods to ROS2, which are then accompanied with innovated procedures that make use of Alloy towards the verification of OD. The implementation of svROS is guided by the following metrics:

- Translation and parsing procedures must be hidden from the developer.
- The generated files must be easy to understand, as it is meant for being used by robotic developers.
- The Alloy models must capture the essence of the application architecture, while being as much as abstract as possible.
- The analysis must be able to capture both structural and behavioural aspects automatically, by deriving them from the generated mega-model.

An overview of the svROS workflow is shown in Figure 24. It starts by implementing procedures for information extraction from application packages, which represent how ROS sets applications from. Extracting (command *Extract*) allows the creation of svROS projects, which are then subject to other command functionalities, namely launching (command *Launch*) and analysing (command *Analyze*).

A user-provided YML project file (Figure 24) instructs the tool to identify a Project (accompanied by a project name), which defines the analysis target. A *project* is the result of information extraction from a ROS application. svROS keeps track of the projects by creating a shared directory, which along with meta-information (which includes a meta-model in Alloy) allows ROS system applications to be set as svROS projects, with an individual sub-directory for each project.

A project file presents a list of launch-files (which can be either in Python or XML), with the primary purpose of identifying the ROS application prone to analysis. This parameterisation through a list of launch-files is necessary because ROS does not provide an explicit mechanism to identify an application. Automated source code fetching comes next. Packages, listed in the launch-files, contain information about executables in configuration files, which represent a node source code, written either in Python or C++. An automatic extraction procedure builds the network topology of the application, requiring minimal effort and expertise. The network topology represents how nodes are connected with each other in the network. Node topic calls, which are specified in the source code of a node, are also extracted to allow the creation of such connections (establishing a publish-subscribe connection).

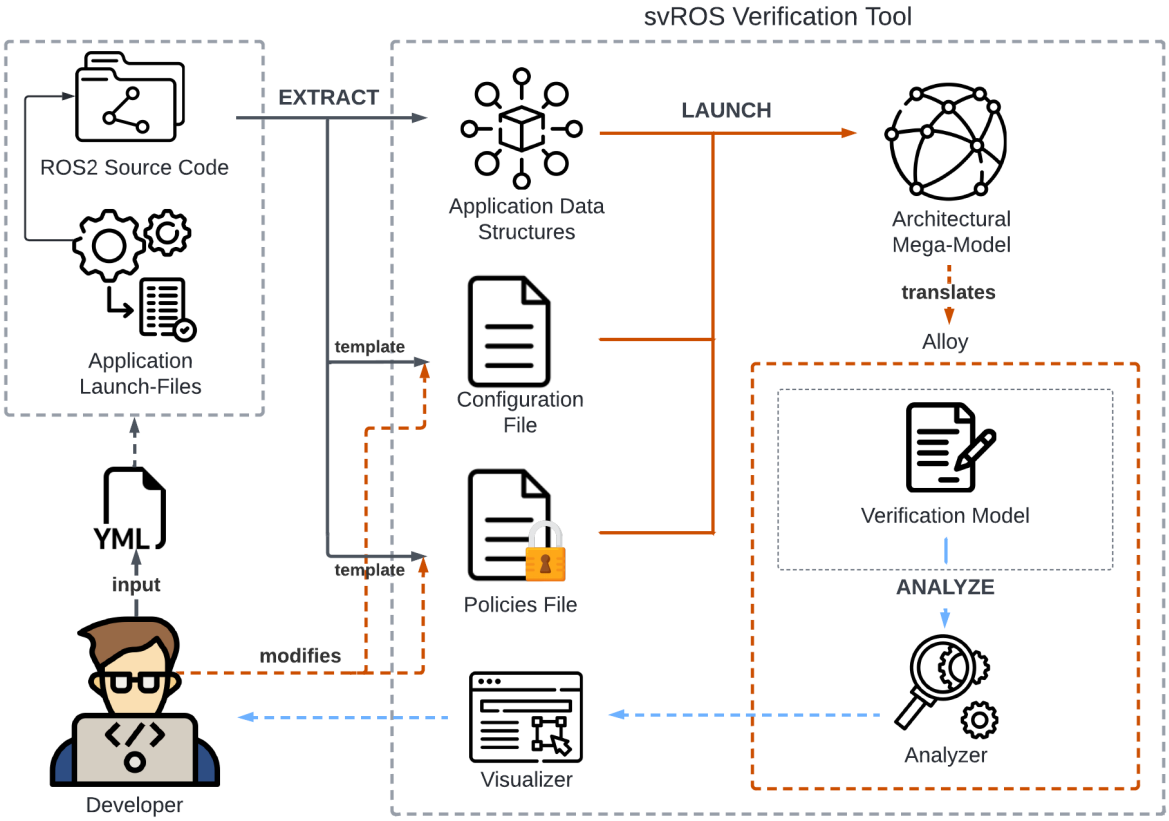


Figure 24: The svROS Workflow Architecture. Capitalised labels are the different execution commands provided by the tool. *Extract* interprets ROS packages and source code, and parses into data-structures and data-files. These are stored in a project directory, alongside with configuration templates. *Launch* builds the architectural *mega-model*, that results from merging all the required information from both the configuration file and the policies file. Through translation procedures, Alloy templates are generated with regard to the architectural mega-model. These models are verified using the *Analyze* command, that implicitly makes use of the Alloy Analyzer. If any counterexample emerges, then svROS Visualizer is capable of parsing the given results and displayed them using a web-interface.

Figure 25 depicts an overview over how svROS performs this basic static analysis (using the *Extract* command), where source code information is extracted without requiring prior compilation. The information is distributed across multiple data files, and is temporarily stored in data structures and objects, inside a project directory. The most helpful data file is written in JSON, which describes the network topology and entity (namely, node and topic) information, in a readable format.

The extraction procedure makes use of existing techniques for static analysis of ROS applications, namely the work of Santos et al. in **High-Assurance Robot Operating System (HAROS)** [56, 57, 58, 59]: ROS package finding, source code fetching and launch-file parsing. svROS extends such functionalities to incorporate the API of ROS2 and its documentation format, since HAROS is unable to cover them due to ROS and ROS2 file format incompatibility.

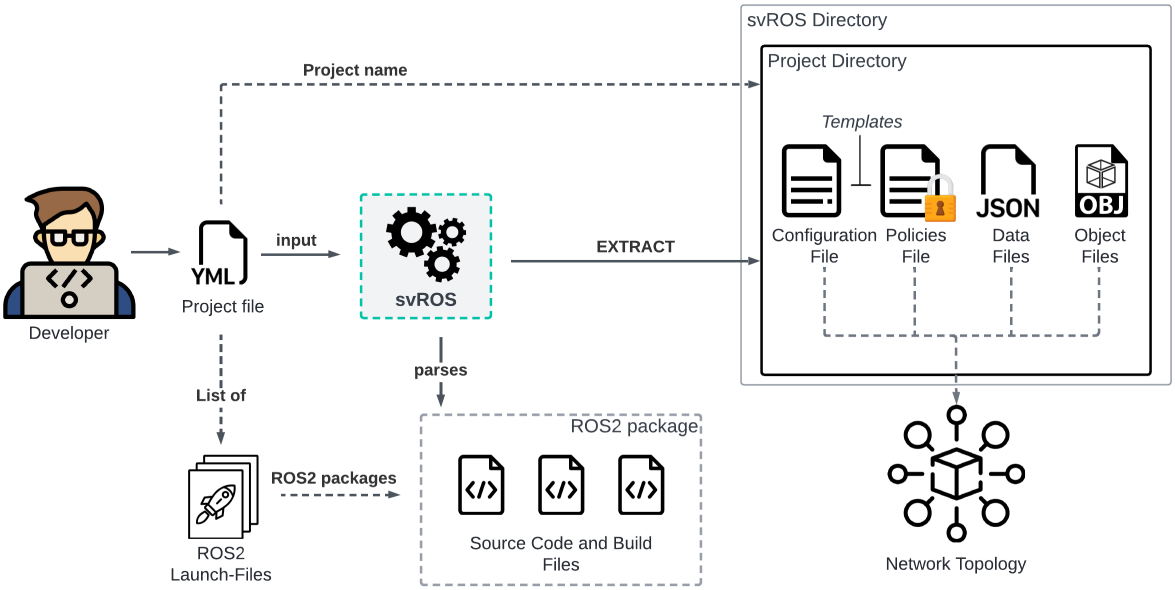


Figure 25: Workflow of the *Extract* command.

Information extraction is complemented with the creation of predefined configuration templates, which can later be modified by the developer. The Configuration File, written in YML, represents a draft of the application architecture, including nodes and packages; and the Policies File, where nodes across the network are configured with privileges of access. The last one is obviously written in XML, with respect to the SROS2 syntax format. This file is created upon the network topology as a draft, where if a node has a subscribing topic call, a privilege of subscribing to that topic is instantiated in its profile. The generation of these templates accounts how launch-files might give information about nodes association with enclaves: if any node n executable gives as a launch argument an enclave path en, the configuration file associates en as the enclave of n, while in the policies file, the en path is created with the node's path n as a profile

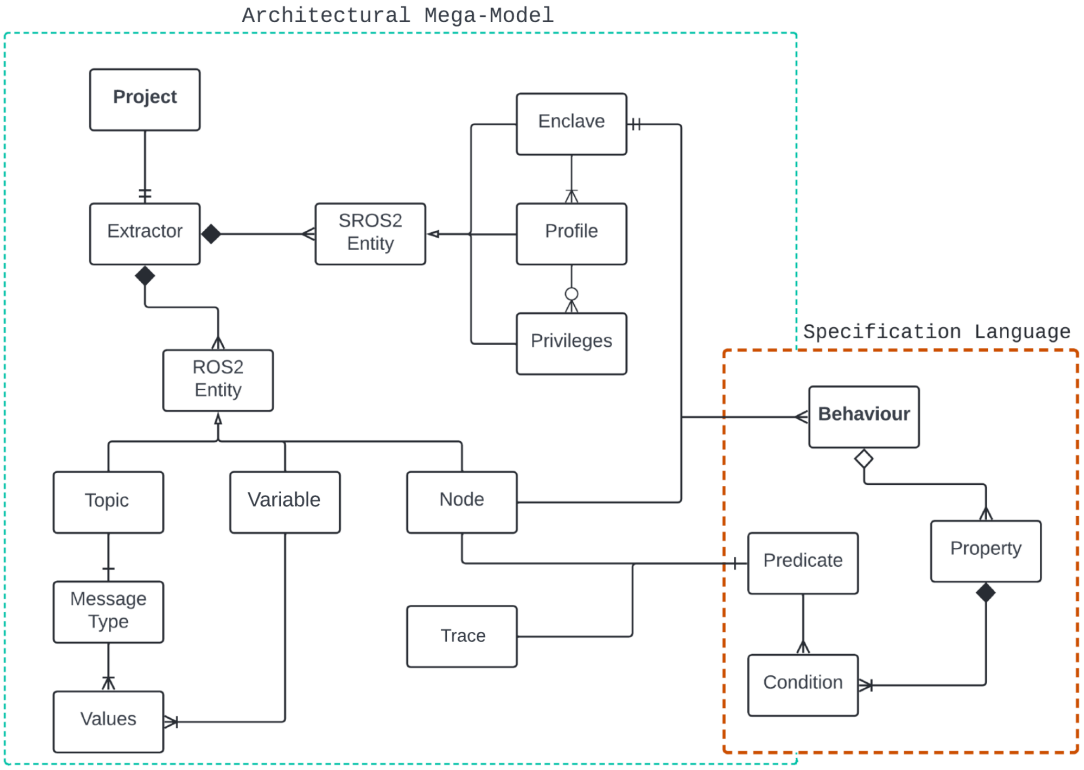


Figure 26: The architectural mega-model class diagram.

of that enclave. Any node without an enclave is instantiated in a predefined enclave path, identified as *public*, which is later set as the public (untrusted) enclave. Again, the developer is free to change these generated templates.

Project launching is the most advanced functionality of svROS. The *Launch* command takes a project name and interprets the configuration files in its directory. These files, which once were mere templates, should by now be altered by the developer. The configuration file should contain information about packages and nodes, whereas the policies file contains information about enclaves, profiles, and privileges of access.

Nodes in the configuration file are matched to a profile defined in its enclave. Profile privileges denote the allowed topic connections between nodes through either publishing or subscribing permissions. This enables svROS to infer an architectural mega-model, linking nodes and topics similar to the network topology, but now with additional information to support later verification methods. Figure 26 shows the mega-model class diagram with regard to its named source entities.

Verification of OD requires sophisticated model-checking methods that use the configuration of security

aspects across the network. Such methods rely on the use of Alloy, while the architectural mega-model is used as the main source of information. Besides the system application structure, namely its nodes and topics, to perform such analysis, the mega-model must also contain:

- The application initial assumptions that constrain the system.
- The specification of intra-node behaviour.
- An indication of which entities are prone to security analysis (untrusted entities).
- The application variables that interfere on how the system behaves, not captured by nodes or topics.
- Verification scopes for Alloy.

The final goal is to create a convenient bridge between the ROS context and formal approaches, enabling a robotic developer to make use of svROS to perform a security analysis, namely the verification of OD. This requires individual node behaviour specifications, where the technique parses and formalises into verification models.

Using Alloy to specify behaviour is not ideal, as most developers are not familiar with such language. This concern led to the development of a **Domain Specific Language (DSL)**, titled svROS specification language, that is expressively enough to specify ROS node behaviour. It centres its specification core on the publish-subscribe paradigm, which represents how nodes interact within a network.

DSL specifications are relative to nodes from the configuration file: the identification of a node is accompanied by in-line DSL properties, which define its intra-node behaviour. This means that DSL specifications are parsed upon launching the project, particularly when interpreting the project's configuration file. Figure 26 also shows how the architectural mega-model maintains the connection between a node structure and a set of behaviours.

A ROS application can have specification variables, usually written in node internal code as programming variables, whose value can play a role on how the system behaves. The developer can identify any application variable in the configuration file, and then use them as behavioural variables in the DSL node specifications. In addition, these can be set as either public or private, which interferes with how the public state of the system is inferred, and which nodes can access that variable (a private variable can only be accessed by private nodes).

Another feature is the specification of the initial assumptions of the application. Initial assumptions are used to constrain verification models on what the developer initially expects the system to be. The

developer can use the configuration model to specify which conditions must hold in the initial state of the verification models.

Lastly, verification models in Alloy are verified upon scopes. Most of the scopes are automatically covered by scopes on signatures, but others must be defined by the developer, since their value is extremely relevant to the verification universe. These are the size of the topics' buffer and the number of verification steps, which are both set by the developer in the configuration file.

Figure 27 shows the full functionality of launching a project, using *Launch* execution command: from interpreting the configuration files, up to translating structures from the mega-model to verification models in Alloy.

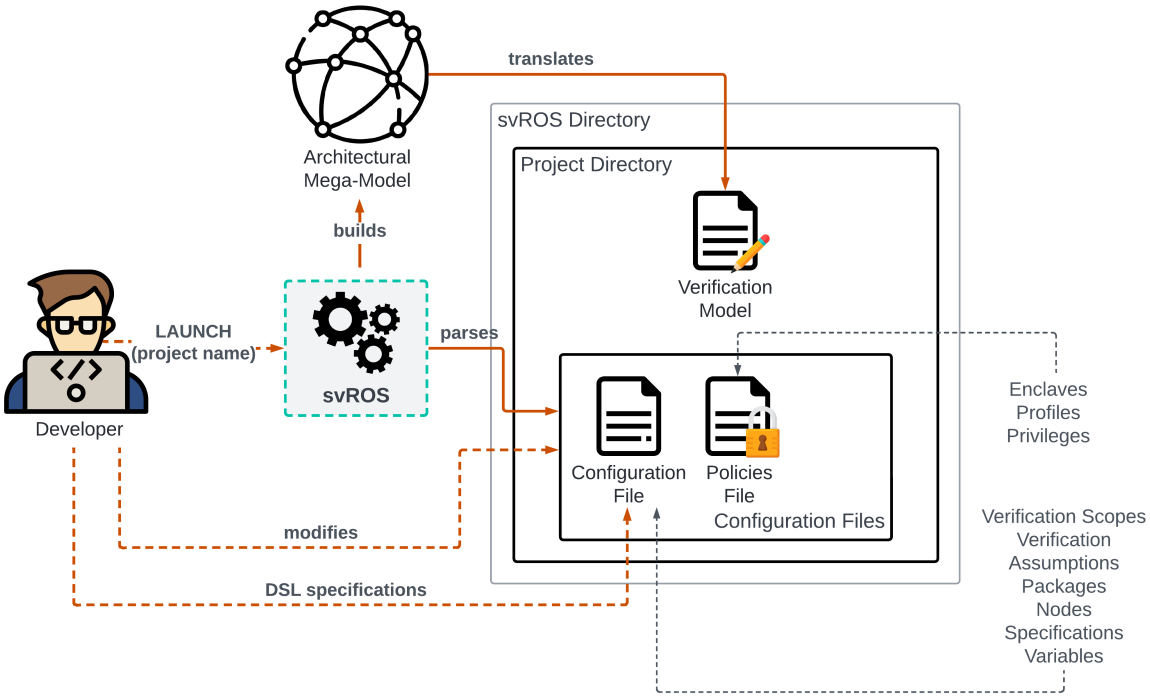


Figure 27: Workflow of the *Launch* command.

Lastly, the *Analyze* command instructs the tool to perform model-checking in the project's verification model, as shown in Figure 28. This mechanism was decoupled from launching to allow a developer to modify the model freely and, most importantly, to comprehend how the translation was made.

Upon running the *Analyze* command in a project, the Alloy Analyzer captures possible counterexamples on the verification of OD. These are temporally stored, to then be consequently parsed and displayed by the Visualizer.

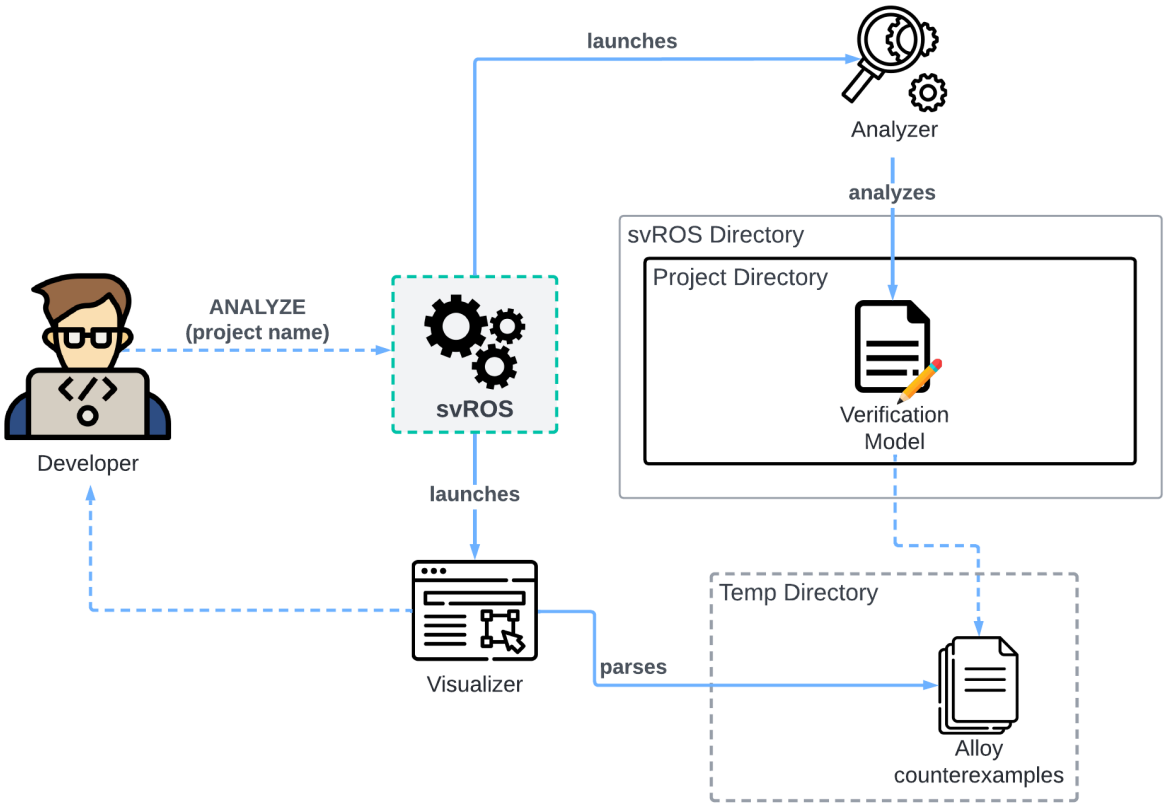


Figure 28: Workflow of the *Analyze* command.

The svROS Visualizer enjoys a web interface to present counterexamples automatically, while being visually more understandable than Alloy's visual interface. It is also capable of showcasing the network topology of the system, through the implicit use of the architectural mega-model that determines linking between nodes and topics and how enclaves enclose nodes. This functionality is presented upon running the *Launch* command, where the developer can opt to open the Visualizer to view the network topology.

5.2 Verification Models for ROS

In ROS, every application makes use of distinct nodes and topics, and consequently behaves differently. However, their specification can be obtained by refining a common meta-model that captures ROS topic behaviour. svROS builds verification models of an application by extending a meta-model that contains the basis of every ROS architecture. It automatically infers declarations, constraints, and assertions from the architectural mega-model, towards the right specification of an application, to then perform verification of OD. For this, it must also infer Alloy constructs to implement self-composition and specify public-event synchronisation.

Self-composition can be done almost entirely independent of each concrete application. The association between topics and a sequence of messages can be modelled independently of the inferred topics, which are exclusive to the application. Variables, however, are user-instantiated and are application dependent. svROS is capable of parsing and transforming each variable into an execution-dependent field.

OD relies on the overall security evaluation of the architecture of the application. Entities (namely, nodes, topics and variables) are classified as either public or private. This classification allows svROS to specify synchronisation between public events and verify equality between observable events.

5.2.1 Verification Model

The meta-model (Listing 5.1) captures the structural basis of an application in ROS. The structure denotes the application architecture: identification of which nodes and topics are used, and their publish-subscribe relation.

```
abstract sig Node {
  subscribes, advertises : set Topic
}
abstract sig Not_Numeric {}
sig Message = Not_Numeric + Int {}
abstract sig Topic {}
```

Listing 5.1: Excerpt of the Alloy ROS2 meta-model.

Modelling a concrete application is done by instantiating the abstract signatures of the meta-model. Each application might have distinct configurations; therefore, each configuration yields a distinct model. Figure 29 illustrates the instantiating of the common meta-model into a concrete verification model.

It makes use of the architectural mega-model to build the verification model: nodes and topics are structurally related in the latter, which enables the declaration of the `subscribes` and `advertises` relation fields.

Message Values

Message values are key to the verification model since they are relevant to the run-time behaviour in a publish-subscribe architecture, since a node's behaviour is expressed as relations between input and output message values, implicitly handled by topics.

In ROS, topics have a corresponding type that the respective messages should respect. Generally, every ROS data-type can be abstractly mapped into two different main type categories: numerical and non

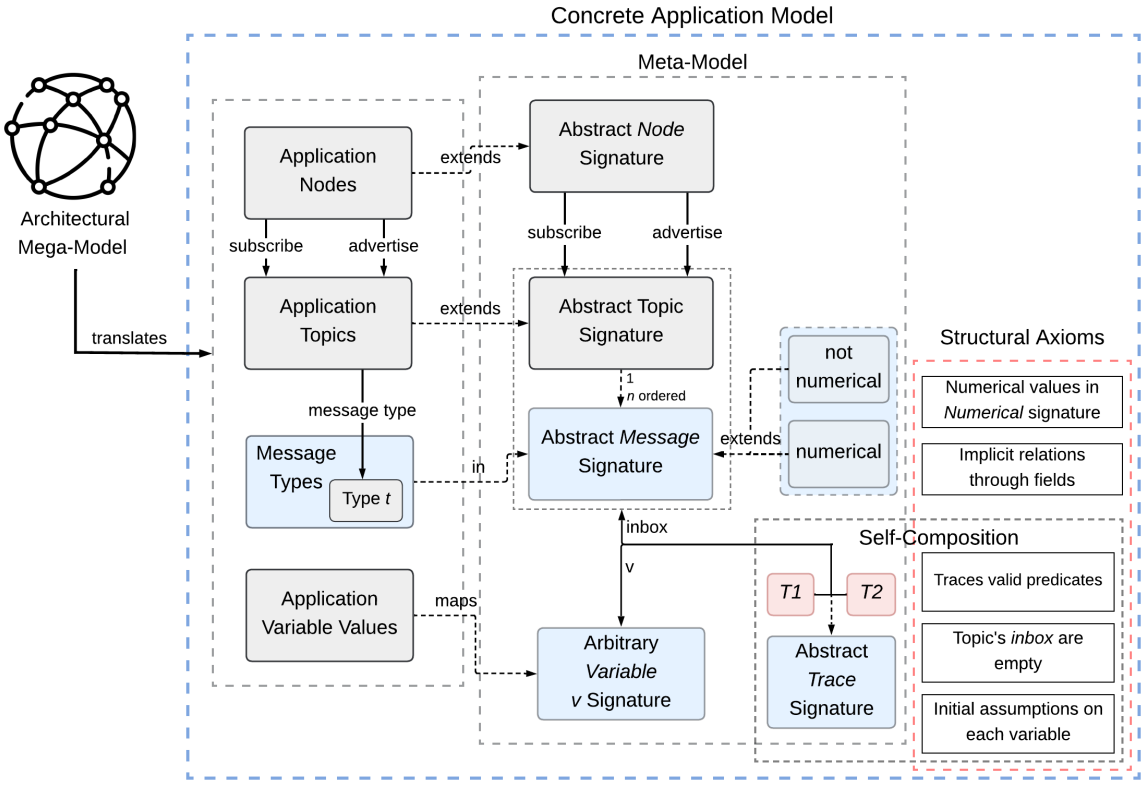


Figure 29: Representation of the main components within every generated Alloy model. Solid connections illustrate static relations between signatures, apart from the relations originating from the two self-composed traces ($T1$ and $T2$), which are time-variable. Dotted connections represent hierarchical relationships.

numerical. This distinction allows svROS to make use of arithmetic operations for numerical types, which enables a more precise specification of how a node behaves. Non numerical types include enumerated data types that do not require arithmetic operations to be processed. For those, only equality is supported.

Type values are inferred from the intra-node specification, which means that the scope of type values is not user-provided. In a non numerical data type, if a behaviour specification uses a value v of a message read or published in a topic, svROS automatically creates a singleton signature for it, which extends the *Not_Numeric* signature. On the other hand, numerical signatures behave differently, and most of the times the bound of numeric values is static.

The modelling of numerical values makes use of Alloy's *Int* signature. The downside to this is that the integer range of *Int* is very short by default (from *-8* to *7*), which might not be expressive enough to represent every relevant numerical value. Increasing the *Int* scope by one duplicates this range, which considerably reduces verification performance.

In [12], the authors present an approach in which numerical values signatures are interpreted as

intervals of values. However, that approach does not support arithmetic operations over values, which is useful for specifying the behaviour of most applications.

A solution for having finer control over the scope of the numerical values could be to make use of *enum* to declare a signature with an orderly enumeration of singleton atoms, with each representing a different number. Ordering allows to perform arithmetical operations through the use of the *prev* (subtraction) and *next* (addition) ordering predicates. This implementation would be a possible add-on to svROS; however, for the moment, the tool uses the *Int* signature.

5.2.2 Self-Composition

Time-variable parts of the application are what self-composition duplicates. Topics, in their essence, are not time-variable, since they represent a communication bus between node ends; however, their message buffer is. Message buffers can receive a message and then proceed to broadcast it, which removes it from the buffer. Therefore, self-composition captures how topics are related to their message buffer through the *inbox* field relation.

Application variables are also set as time-variable, due to the fact that they interfere with how the system behaves. Their evaluation on a particular state plays a role on the verification outcome of the analysis.

The same approach on self-composition explained in Chapter 4 is the one used by svROS to set how system traces are duplicated. An abstract *Trace* signature is declared, which is extended by singleton disjoint signatures $T1$ and $T2$. This enables Alloy to evaluate these execution traces separately. Time-variable parts are captured by relation fields that duplicate the topics' message buffer (their inbox) and each application variable.

Listing 5.2 depicts an example on how self-composition is handled in the Alloy verification model. The inbox is modelled independently of the application. Variables are application-dependent, and in this particular example, a variable vr was previously instantiated and translated as time-variable field relation of the signature *Trace*.

```
abstract sig Trace {
  var vr   : one Value,
  var inbox : Topic -> seq Message
}
one sig T1, T2 extends Trace {}
```

Listing 5.2: Declaration of traces in the verification model of an application, including the two trace replicas $T1$ and $T2$.

Trace evaluation is constrained using an axiom fact to determine which events can happen during the evolution of a trace, also accounting stuttering as a possibility. An event characterises how the trace evolves in-between two states and is specified by an Alloy predicate.

A trace is taken as an argument for each event predicate, as shown in Listing 5.3: one is the stuttering event (or *stutter*), and the other is the concrete system predicate. The latter captures the whole system behaviour, and is instantiated with predicates describing atomic intra-node behaviour, resulting from the translation between the DSL specifications and the Alloy language. Each node is specified by a predicate to ease the understanding of traces, as it is clear to infer which node executed between consecutive states.

```
fact system_behaviour {
  always ( stutter[T1] or system[T2] )
  always ( stutter[T1] or system[T2] )
}
```

Listing 5.3: Events of the system: traces $T1$ and $T2$ can either evolve through a system behaviour (the behaviour of nodes) or through stuttering.

Ordering in Topics Verification involving communication and message handling is challenging. Models should represent message-passing in an abstract manner; however, in some systems, the order of message delivery is relevant. That is the case of ROS topics. They act as n-sized buffer of messages between nodes, storing multiple messages that are then handled with a *FIFO* policy. This requires ordering of the messages in the relation *inbox*.

Fortunately, Alloy supports a special multiplicity keyword, namely **seq**, which can be used as an alternative to storing messages in a set and then imposing an order on the respective atoms. The *seq* adding operation drops elements when it is full, such as topics when the message buffer reaches its maximum capacity.

The only thing worth noting is the scope of *seq* corresponds to the size of topics. The *seq* scope can be set upon analysis. The downside to this is that different-sized topics are not supported, because by setting *seq* with a scope value, it declares the size for every topic. This declaration also has an entire set of predefined predicates and functions, making it easy to get and update any message buffer.

Variable Values Variable values follow the same path as message types: numerical or non numerical. Non numerical variable values are assigned exclusively to a variable, which means variables do not share the same scope of values. Similarly, values are inferred from the node DSL specifications, or from initialisation assumptions.

Replication of Inboxes Storing messages in topics exclusively does not correctly expresses how topic connections between nodes work in ROS. A topic inbox deletes a message after sending it to all the subscribing nodes. However, our verification model does not represent node inboxes. A message is deleted from a topic when one of its subscribing nodes reads it.

If two different nodes subscribe to the same topic, in ROS they would receive the same message in their inbox. As it is, however, if one reads the message prior to the other, the latter never receives it. One solution to this is to modify the model to add inboxes in nodes. However, this would lead to other modelling concerns: messages would have to be distinguished depending on which topic they were sent, and it would also negatively affect analysis efficiency. An alternative is to replicate a topic for every possible subscriber.

A topic c that is subscribed by n nodes would have n inbox replicas, and when a publishing through it occurs, the same message would be added to these replicas. This replication is abstracted by the tool, since only the verification model makes use of this solution.

5.2.3 Observational Determinism

Communication establishment solely relies on how privileges are distributed across node profiles in the architectural mega-model. Again, the latter identifies these privileges association with nodes mostly out of the policies file.

Through this, svROS is capable of building communication data-structures, linking nodes through publish-subscribe relations. Additionally, it also identifies which publishing events should be affected by the synchronisation of public events. Lastly, svROS automatically generates assert specifications to verify OD on public observations.

Public-State Equivalence

Firstly, a fact must enforce the initial public-state equivalence of traces. Remember that this is a condition necessary to verify OD. The public state of the application system encompasses public variables and public topics. However, there is a slight difference in how svROS treats their equivalence:

- Public variables have initially the same value in both traces.
- Topic inboxes start empty of messages in both traces.

Naturally, variables defined as public start equally valued between traces, whereas, the ones set as private have no restrictions imposed over them.

Topics, likewise variables, can be classified as either private or public. Private topics do not contribute to public-state equivalence; however, svROS sets every inbox as initially empty. This obviously represents a strong refinement to the definition of public-state equivalence, but it leaves most of the nondeterministic behaviour only to trace transitions, which is far more reasonable.

Listing 5.4 shows how svROS imposes public-state equivalence on the initial state of the system declared in Listing 5.2, assuming that the application variable vr is set as public.

```
fact {
        T1.vr = T2.vr
        no inbox
    }
```

Listing 5.4: Initial state constraints on a system application. The first constraint imposes that public variable vr has the same initial value in both traces. The second states that inboxes start empty.

Public-Event Synchronisation

Events are characterised as publishing occurrences through topics. A public event is a publishing occurrence through topics set as public, where these are identified as follows:

- Topics advertised by any public node.
- Topics advertised by private nodes, but for which privileges allow public nodes to subscribe to it.

Considering this definition, svROS generates an axiom whose formula enforces the lock-step synchronisation of publishing in topics set as public. Examples of such topics are depicted in Figure 30.

Besides enforcing the publishing synchronisation, svROS discerns public topics advertised by public nodes from those advertised by private nodes. That is because publishing from public nodes represents an input to the system. So, when a public node publishes through a public topic, the synchronisation axiom forces the same message to be sent in both trace replicas.

Listing 5.5 shows how the translation handles event synchronisation in such topics, using the examples of Figure 30, where only topic $c1$ handles messages published by a public node a.

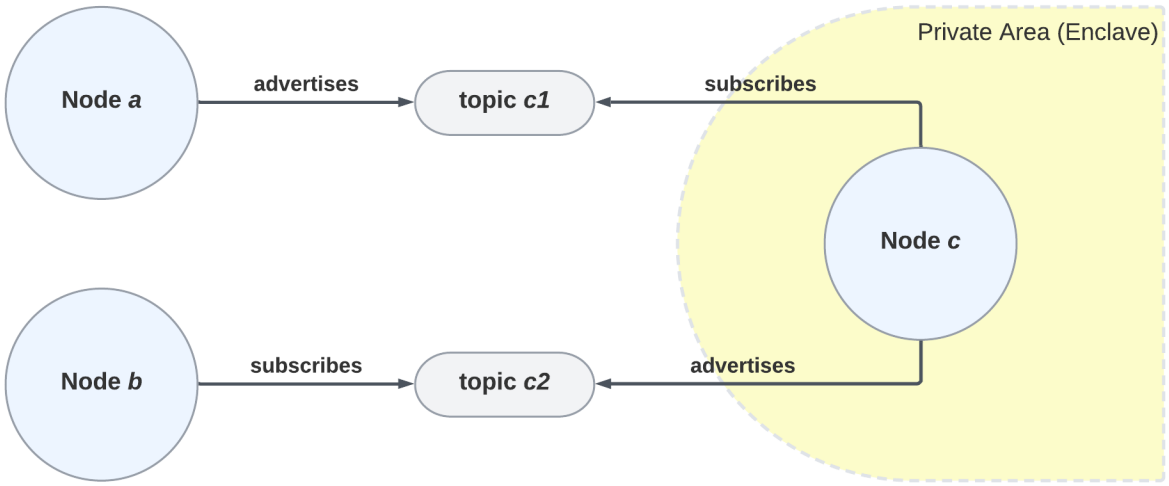


Figure 30: Examples of *public topics*.

```
fact {
        always (
            all m : Message | publish[T1, c1, m] iff publish[T2, c1, m]
        )
}
```

Listing 5.5: Public-event synchronisation in public topic $c1$.

Public topics advertised by private nodes are labelled as public observations. OD is verified upon these observations, namely by checking how deterministic these observations appear to untrusted nodes. In Figure 30, public topic $c2$ handles messages issued by a private node c, from which public node b can read.

Public observations, as public events, should also be trace synchronous. However, its specification should not assume message equality, as in topic $c1$, because the equality of the sent message is precisely what OD verifies.

Listing 5.6 shows how the lock-step synchronisation of these topics is handled by svROS. Messages m and n are not necessarily equal, only the act of publishing is synchronous.

```
fact {
        always (
            ( some m : Message | publish[T1, c2, m] )
            iff
            ( some n : Message | publish[T2, c2, n] )
        )
```

```
}
```

Listing 5.6: Public-Event synchronisation in public topic $c2$.

Verification of Observational Determinism

OD concerns the public observations of the system. svROS automatically generates assertions to verify equivalence on these observations. svROS follows the same path of public-event synchronisation on public observations, but instead of translating equality as an assumption, it creates a verification assert.

The assertion in Listing 5.7 checks whether node c publishes the same message, through public topic $c2$, in each trace replica. For OD to hold, messages m and n must necessarily be the same.

```
check {
        always (
            all m, n : Message | publish[T1, c2, m] and publish[T2, c2, n]
                                 implies m = n
        )
}
```

Listing 5.7: Assertion to check OD in topic $c2$.

5.3 Specification Language

A specification language that fits software development in ROS must prioritise communication and message-passing. Nodes act upon messages, whose data is very relevant to influence a node's behaviour. Thus, a specification language for ROS must rely on the relations between messages and, considering how messages work, it must also feature the following:

- References to communication entities.
- References to messages and their data.
- Relations between publish-subscribe calls.

svROS specification language is a user-oriented and context-specific DSL because it references concepts that are naturally familiar to ROS developers. It offers a simple and easily understandable specification core based on publish-subscribe patterns.

Specifications are restricted by the language's syntax, enclosing patterns that improve its usability. Figure 31 shows the syntax whose semantics operate at the level of message passing, treating each corresponding node as a black-box. A node N is specified by a sequence of patterns, whose informal semantics are as follows:

- **Read** pattern: Reads the oldest message m of a topic c buffer. N must have a subscribing privilege over c. Node N reads m and can operate over it, either through publishing, further reading, or updating a variable. Any operation contemplates a condition, a *pass statement* (if the condition holds) and a *fail statement* (if the condition fails).

- **Publish** pattern: Publishes through a topic c in the next state. N must have a advertising privilege over c. Publishing values can be arbitrary, if no value is specified. Alternatively, it can be set to a value v or a range of values.

- **Update** pattern: Updates the value of a variable s in the next state. If N is public, and s is private, N cannot update s.

- **Require** pattern: Requires a guard condition $cond$ to be true, which controls when is the node enabled.

A property is conceptually divided following the hierarchical structure depicted in Figure 32. The tool is then capable of creating a direct translation between the language and Alloy using the respective parser and some intermediate data structures.

Each node is specified by a sequence of properties, and the tool automatically creates an Alloy predicate encoding its behaviour following the above informal semantics. Naturally, if no property is specified for a node, the respective Alloy predicate will be equivalent to the stuttering event. The disjunction of all node predicates corresponds to the system behaviour.

Dealing with tokens Managing tokens with the same expression but defined in different contexts can lead to parsing conflicts. This is because the grammar parser used is a *LALR* parser, a bottom-up parser that cannot identify lookahead (or subsequent) symbols. That is the case of tokens such as *topic*, *variable*, and *predicate*. In order to avoid non-recognition conflicts, these tokens are initialised differently, so the tool can recognise them individually: topics are naturally identified as having an initial "$/$" in ROS (owing to their namespace). Variables are identified with a "$\$$" and predicates with a "#".

```
property    := pattern
pattern     := requires conditional
             | event
event       := publishes ( topic [ topics ] )
             | updates   ( variable evaluate )
             | reads     ( topic [ topics ] )
conditional := disjunction ( and conditional )*
disjunction := condition ( or disjunction )*
condition   := [ no ] cond | event
topics      := as message_token { implication }
             | evaluate
implication := operation ( ; operation )*
operation   := if conditional then { conditional } [ { else operation } ]
             | conditional
cond        := param [ evaluate ]
evaluate    := binop ( value | variable )
binop       := = | != | < | > | += | -= | >= | <=
param       := topic | variable | predicate | message_token
```

Figure 31: Syntax of the svROS specification DSL. Bold keywords are language static tokens. Italic keywords are pattern tokens used to identify an entity.

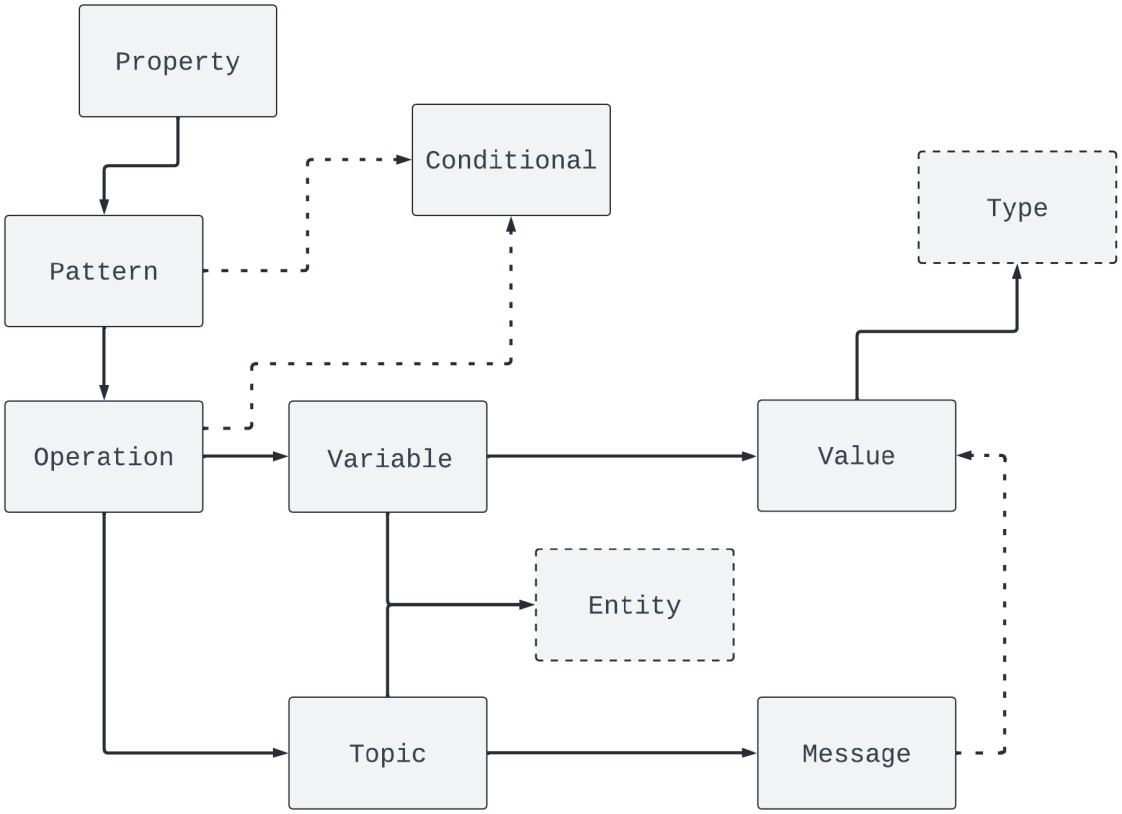


Figure 32: Property structure. Dashed lines represent optional elements. Dashed elements represent hidden elements that are implicitly linked to a property, through the architectural model.

Frame Conditions After parsing each property, the tool separates which topics and variables the node uses and those which not. These are identified as *non-changeable* topics and variables, which in Alloy are translated as frame conditions, forcing them to remain unchanged in the next state.

However, Alloy is very sensible to frame conditions; i.e, an event behaves unexpectedly, if any variable signature or field, that should remain the same, is not specified as a frame condition. A common issue is when a topic is used in different if conditionals. If the conditional does not hold, how can the tool infer whether the topic has changed? For instance, suppose this scenario, where a node *node* reads a number from topic $c1$, and then publishes the value of 1 through topic $c2$, if the number is higher than 0, and publishes the value of 2 also through $c2$, if the number is lower than 0.

```
behaviour:
    - reads /c1 as m { if m > 0 then { publishes /c2 = 1 } ;
                       if m < 0 then { publishes /c2 = 2 }
                     }
```

Listing 5.8: DSL specification of a node with two conditionals in sequence.

What happens if a message is read as equal to 0? Well, if topic $c2$ is set as changeable, $c2$ is free to change in the next state. On the other hand, if topic $c2$ is set as unchangeable, it depends on what value m is equal to: if $m = 0$ this would behave correctly; but, would obviously cause an inconsistency if $m > 0$ or $m < 0$ (that requires a change to the topic in the next state, which would deny the setting of the topic to unchangeable).

The solution was to restrict the usage of a topic to a single operation, meaning that the same topic cannot be used in two operations separated by a semicolon. The developer should only separate by a semicolon operations regarding different entities (either variables or topics). To specify the above behaviour, nested conditionals should be used instead.

```
behaviour:
    - reads /c1 as m { if m > 0 then { publishes /c2 = 1 }
                       else { if m < 0 then { publishes /c2 = 2 } }
                     }
```

Listing 5.9: DSL specification of node with two nested conditionals.

In Alloy, the latest behaviour would produce a fail-statement that encloses the whole if statement, which would capture the case when m is equal to 0.

Sub-predicates A node predicate can be divided in multiple predicates. These are also called *sub-predicates*. The tool tries to find sub-predicates within the node behaviour specification through pattern finding, before parsing in-line properties. Sub-predicates are also composed by DSL properties, that are translated into different Alloy predicates. Since these are not set as top-level predicates, the system behaviour calls only the *main* top-level predicate in its disjunction. In the main predicate, only one of the sub-predicates can be true (they are invoked inside a disjunction Alloy connective *or*). This idea facilitates the specification of the behaviours of conditionals or nondeterministic nodes, such as the multiplexer, which can have different execution patterns depending on shifting factors.

For instance, consider a node *node*, that requires that a variable var is evaluated to True. It also behaves nondeterministically through either sub-predicate *one* or sub-predicate *two*. The former updates var to False, whereas the latter reads a message from topic $c1$ and broadcasts it to topic $c2$. The corresponding DSL specifications are as follows:

```
behaviour:
    - requires $var = True
    - one:
        - updates $var = False
    - two:
        - reads /c1 as m { publishes /c2 = m }
```

Listing 5.10: DSL specification of a node with sub-predicates.

The translation into Alloy results in three different predicates: the main predicate *node* and the two sub-predicates *one* and *two*. Remember that event predicates are applied to trace replicas (they take a trace t as argument) due to self-composition. Considering how *seq* works in Alloy (for instance, to add a message to an inbox, the predicate add is used) and the evaluation of the fields in the next state (computed with ′), the translation is as follows:

```
pred node [t : Trace] {
    t.var = True
    one[t] or two[t]
}
pred one [t : Trace] {
    t.var' = False
}
pred two [t : Trace] {
    some m : t.inbox[c1] {
        t.inbox[c2]' = add[t.inbox[c2], m]
    }
}
```

Listing 5.11: Alloy specification of a node with sub-predicates.

The main downside of this language is that it lacks support for real-time behaviour specifications. Properties that rely on time measurement are not supported because Alloy does not support a notion of real time. Moreover, as the language is not excessively sophisticated, the developer might fail to express some behaviours. For instance, the nesting of read and publish patterns might not be directly matched to ROS's subscribe and publish callbacks.

5.3.1 Initialisation Assumptions

The language was mainly created to be able to express intra-node behaviour. Notwithstanding, the system can have assumptions that characterise how the application initiates. Remember that variables and topic inboxes have direct impact over self-composition and verification of OD. Inboxes start initially empty; therefore, no further assumptions about their initial values are necessary.

Thus, initialisation assumptions are only relevant to variables, with special care to private variables, since if no assumption is made over these, they can differ between the two trace replicas. An initialisation assumption uses the *Require* pattern of Figure 33.

```
property    := pattern
pattern     := requires conditional
conditional := disjunction ( and conditional )*
disjunction := condition ( or disjunction )*
condition   := [ no ] cond
cond        := variable evaluate
evaluate    := binop ( value | variable )
binop       := = | != | < | > | >= | <=
```

Figure 33: Syntax for system initialisation assumptions.

For instance, consider the latest example and the application variable var. As an initialisation assumption, consider that var is initially set to False. Using the DSL for initialisation assumptions, the developer can specify this constraint as:

```
requires $var = False
```

Listing 5.12: Initialisation of the variable *var* in DSL.

The translation of this assumption into Alloy is enclosed in an axiom called *initial_assumptions*. Listing 5.13 captures the resulting svROS translation.

```
fact initial_assumptions {
    T1.var = False and T2.var = False
}
```

Listing 5.13: Initialisation of the variable *var* in Alloy. Remember that any variable is defined within signature *Trace*, so assumptions on $T1$ and $T2$ are considered individually.

Chapter 6

Evaluation

This chapter presents an empirical evaluation of the techniques presented in both Chapter 4 and Chapter 5. Both technique expressiveness and performance are evaluated, supported by the previously mentioned *TurtleSim* ROS2 application.

Evaluation also measures on how precisely can svROS automatically infer an Alloy formalisation from a ROS application. This section shows the result of the translation between the application configuration and behaviour specification, to the Alloy verification model used to perform verification of OD.

The main challenge of any type of Alloy analysis is the performance, usually conditioned by scopes on signatures and steps, owing to its bounded model-checking technique. Thus, the evaluation is conducted for different bounds in steps, while resorting to different solvers. This approach aims to allow the reader to better comprehend possible limitations in verification using Alloy.

The verification of OD mostly relies on the number of steps, the scope of messages and the size of topic inboxes. The scope of messages is automatically inferred from the configuration file. Steps and the size of inboxes, however, must be indicated by the developer in the configuration file.

Concerning the performance evaluation, the verification of OD was done on a 2.6 GHZ Inter Core i7 with 16GB memory running on Arch Linux (5.18).

6.1 Specification

OD asserts are automatically inferred from the application architectural structure, through identification of the public parties and their interaction with private ones. The tool assumes that public parties are enclosed in an enclave named *public*.

Connections in the architectural mega-model are inferred from the policies file. Before launching a project, the developer should be aware on how nodes are matched to the privileges of access. Remember that the nodes listed in the configuration file are matched to an enclave path. Therefore, the latter must

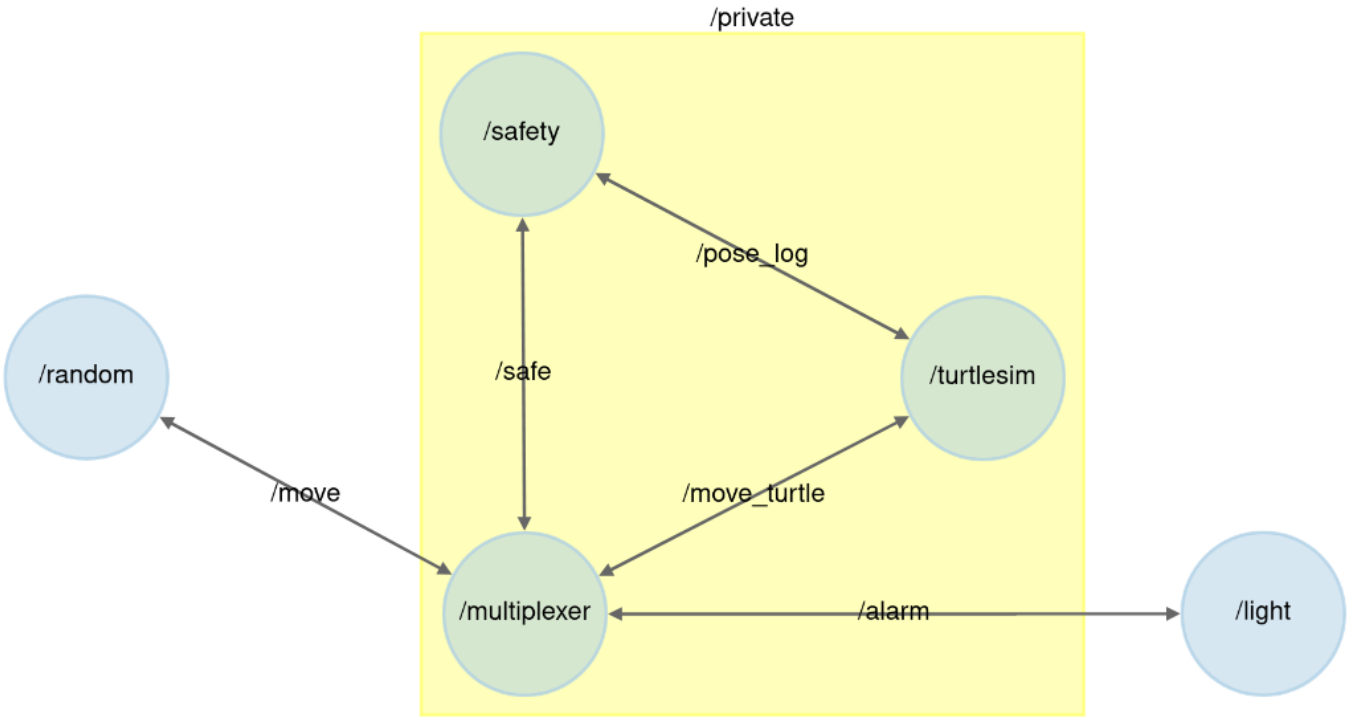


Figure 34: Architecture of the *TurtleSim* depicted by svROS Visualizer.

have a corresponding profile inside the enclave path to be launched in ROS, and most importantly, so that svROS can infer the communication topology.

Listing 6.1 presents the architectural model defined in the configuration file and Listing 6.2 presents the corresponding policies file, where the privileges of access are defined. Figure 34 depicts the network topology of *TurtleSim* using the svROS Visualizer, after the architectural mega-model was built by svROS.

Take for instance two nodes from the configuration file, *Random* and *Multiplexer*, with their respective enclaves. Their enclaves, in the policies file, must define a profile with the same path as the node. Moreover, *Random* has privileges to advertise to *move* and *Multiplexer* can subscribe to it. The mega-model creates a topic connection between these ends through *move*. It then proceeds to identify which topics are classified as public. *Random* belongs to the *public* enclave, which means that *move* is a public topic. So, the specification of public-event synchronisation captures the publishing through *move* as lock-step synchronised in both traces of execution.

Nodes are corresponded with a sequence of DSL properties defined under the `behaviour` keyword tag. Properties are then translated into Alloy specifications. Such specifications are translated into a predicate that corresponds to the node in which they are defined. For instance, *Random* will have a predicate called *random*, in which the translation of its DSL specification is contained.

Application variables are set under the `variables` tag. svROS allows the developer to classify variables

using labels. One of these is the keyword `int`, which is used to define a numeric variable. The turtle position, identified as *position*, is set as numeric. Application variables are private by default, however, the developer can set variables as public using the `public` keyword.

Lastly, assumptions on how the system is initialised can be specified using the DSL for initialisation assumptions. *TurtleSim* requires that the turtle position is initially set to 0. In the configuration file, such assumptions are set under the `configurations` tag, in the `behaviour` section, as Listing 6.1 shows.

```
configurations:
    model:
        steps: 15
        inbox: 5
        behaviour:
            - requires $position = 0
nodes:
    turtlesim/multiplexer:
        rosname: /multiplexer
        enclave: /private
        behaviour:
            - requires no /move_turtle and no /pose_log
            - low  :
                - requires /move and no /safe
                - reads /move as m { publishes /move_turtle = m ;
                                     if m = -2 or m = 2
                                     then { publishes /alarm = True }
                                     else { publishes /alarm = False }
                                   }
            - high :
                - requires /safe
                - reads /safe as m { publishes /move_turtle = m ;
                                     if m = -2 or m = 2
                                     then { publishes /alarm = True }
                                     else { publishes /alarm = False }
                                   }
    turtlesim/safety:
        rosname: /safety
        enclave: /private
        behaviour:
            - reads /pose_log as m { if m = -2
                                     then { publishes /safe = 1 }
```

```
                                    else {
                                        if m = 2
                                        then { publishes /safe = -1 }
                                          }
                                    }
    turtlesim/turtlesim:
        rosname: /turtlesim
        enclave: /private
        behaviour:
            - requires /move_turtle
            - reads /move_turtle as m { updates $position += m }
            - publishes /pose_log = $position
    public/random:
        rosname: /random
        enclave: /public
        behaviour:
            - publishes /move as m { m >= -2 and m <= 2 }
    public/light:
        rosname: /light
        enclave: /public
        behaviour:
            - reads /alarm
variables:
    int position
```

Listing 6.1: *TurtleSim* configuration file, namely "config.yml". Verification scopes and initial assumptions are set under the *configurations* tag. Node behaviour is specified under the *behaviour* tag, on each node. Application variables are declared under the *variables* tag, which can be labelled as integer (**int** keyword) and as public (public keyword). Only private nodes can operate over private variables.

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

```
        </topics>
        <topics subscribe="ALLOW" >
          <topic>safe</topic>
          <topic>move</topic>
        </topics>
      </profile>
      <profile ns="/" node="turtlesim">
        <topics publish="ALLOW" >
          <topic>pose_log</topic>
        </topics>
        <topics subscribe="ALLOW" >
          <topic>move_turtle</topic>
        </topics>
      </profile>
      <profile ns="/" node="safety">
        <topics publish="ALLOW" >
          <topic>safe</topic>
        </topics>
        <topics subscribe="ALLOW" >
          <topic>pose_log</topic>
        </topics>
      </profile>
    </profiles>
  </enclave>
  <enclave path="/public">
    <profiles>
      <profile ns="/" node="random">
        <topics publish="ALLOW">
          <topic>move</topic>
        </topics>
      </profile>
      <profile ns="/" node="light">
        <topics subscribe="ALLOW">
          <topic>alarm</topic>
        </topics>
      </profile>
    </profiles>
  </enclave>
</enclaves>
```

```
</policy>
```

Listing 6.2: *TurtleSim* SROS2 policies file, namely "policies.xml". Here each node's privileges of access is set, using the default syntax of SROS2.

Wrong node specifications can lead to false verification results. This results from the fact that the developer has full responsibility over the specification of these files, which requires great care.

Even though certain behaviours are somewhat complicated, for instance the *Multiplexer*'s, the DSL was expressive enough to specify these. The implicit abstraction in messages might lead to ambiguous specifications and, consequently to false results. However, as far as this study-case goes, this is not a problem.

6.2 Verification

In order to achieve a good confidence in the analysis, the number of steps must be high enough to possibly reach a counterexample. The developer is in control of providing the number of steps through the configuration file within the `steps` tag. Remember that the *seq* scope corresponds to the topics' buffer size. The developer can also provide the size of the latter using the `inbox` keyword tag. Listing 6.1 also allows this under the `configurations` tag. Here, the scopes for steps was *15*, and for the inboxes was *5*.

Remember that the conditions necessary to perform OD verification, namely public-state equivalence and public-event synchronisation, are automatically inferred after the identification of which nodes are public and the topics with which they interact. Public-state equivalence is defined on public variables, of which this example has none. Public events are characterised as publishing occurrences in public topics (in this example public topics are *move* and *alarm*).

Then, the verification of OD checks the equivalence in public observations. These account for private data that is sent from the inside of a private enclave to public parties (namely data published in *alarm*). svROS is capable of generating assertions that check whether observations have equal output values in each trace. Remember that the behaviour being checked here concerns the output through the topic *alarm*.

Figure 35, Figure 36, Figure 37, Figure 38, and Figure 39 in Appendix A depict the verification model that results from the translation made by svROS. It is defined over the common meta-model, considering the information in the architectural mega-model and the DSL specification of the nodes. Each node's behaviour is translated and instantiated in its own node predicate, as shown in Figure 36 and Figure 37.

Figure 38 shows how self-composition is inferred. Lastly, all the necessary conditions to verify OD are presented in Figure 39.

This property was verified under different assumptions. The first specification does not consider the guards in the node *Multiplexer*, that prevents its execution while there are messages to be processed in topics *move_turtle* and *pose_log*. The second specification already considers these guards, meaning that *Multiplexer* is somehow synchronous with these nodes. Thus, the following evaluation considers these slightly different models:

- Model M_1: *Multiplexer* is not synchronous.
- Model M_2: *Multiplexer* is synchronous.

The assertion of OD was checked using increasing scopes for steps. Due to performance reasons, the latter was bounded up to the maximum number of **15** state transactions, which corresponds to the scope given in the configuration file. We believe that the model specification cannot be improved in terms of performance, so we decided to use different solvers in this verification, namely SAT4J, MiniSat, and Glucose. Table 1 shows the verification results for these different solvers, using **exactly** n steps.

Solver	**Configuration**	*n Steps*		
		5	*10*	*15*
SAT4J	**M1**	✓ (4.4s)	✗ (6.2s)	✗ (43.4s)
	M2	✓ (3.7s)	✓ (146.0s)	✓ (1582.1s)
MiniSat	**M1**	✓ (4.1s)	✗ (5.9s)	✗ (10.6s)
	M2	✓ (3.4s)	✓ (16.4s)	✓ (263.6s)
Glucose	**M1**	✓ (3.4s)	✗ (6.3s)	✗ (14.6s)
	M2	✓ (3.1s)	✓ (16.0s)	✓ (239.1s)

Table 1: Results obtained from checking OD public-observation equality on topic alarm, accompanied by their corresponding execution times, using different solvers. Topics' *seq* scope was fixed to *5*.

As expected, model M_1 failed to verify OD, since no process synchronisation within the enclave was considered. The corresponding counterexample emerged within **10** steps of execution. This is the same counterexample that was presented in Chapter 4. Figure 40 to Figure 46 in Appendix B present the evolution of this counterexample now using the svROS Visualizer. On the other hand, the verification of OD in M_2 does not produce any counterexample, even considering **15** steps of execution.

The times measured during this analysis provide empirical evidence of the applicability and utility of the technique. Even though it is safer to perform verification with a high number of steps of execution, the execution time increases rapidly with the number of steps. For instance, using SAT4J with **15** steps already took around 25 minutes in the configuration model **M2**. That is because Alloy is much faster at detecting counterexamples, as can be seen in comparison to the time obtained in the configuration **M1**, which it took less than 1 minute. MiniSat and Glucose greatly outperformed SAT4J, even with **15** steps, their verification never took more than 5 minutes.

The developer is responsible for setting scope values; therefore, the latter must be aware of the communication architecture behind the application. Small scopes will bound verification to small universes, which might not be sufficient to obtain a counterexample. Loose behavioural specifications can also lead to false counterexamples; therefore, the developer must be capable of specifying each node's behaviour without being too ambiguous.

To conclude, this technique was shown to be sufficiently efficient to analyse small case-studies, such as *TurtleSim*. The verification of realistic complete systems would increase the number of event predicates and, consequently, the number of execution steps needed to possibly reach a counterexample. Even though it is still possible to verify OD in such systems, this will most likely not scale. As such, verification should be confined only to the core parts of the architecture that are relevant for security.

Chapter 7

Conclusion

This thesis has presented svROS, a verification tool designed to perform analysis on ROS2 system applications, while accounting for the policy distribution of SROS2. It shows how these can be used to verify OD, through property verification. It requires a configuration file specifying on how the application is structured, a SROS2 file to set privileges upon nodes, and node specifications using the DSL. It automatically creates verification models in Alloy, including the assumptions and assertions needed for the verification of OD. Models are then automatically verified and counterexamples are presented in a more understandable format using the svROS Visualizer.

svROS includes a specification DSL that allows the developer to express intra-node behaviour through properties based on publish-subscribe patterns. This language has proven to be sufficient to describe common behavioural specifications, without the need to resort to the Alloy language directly. The tool has proven to be suitable for checking OD in the *TurtleSim* academic study-case, which represents a first step towards svROS application to real industrial cases.

Real robotic systems are usually composed by a high number of nodes. However, most of them are deterministic, owing to the requirements of real-time computing in such systems. Therefore, enforcing OD might not be as overwhelming as expected. If a system fails to hold OD, solutions will mostly consist of enforcing some sort of synchronisation in nondeterministic nodes.

The main reason that led to exclusively study OD is that it can be formalised as a hyperproperty without alternating different quantifiers. Using Alloy to specify other hyperproperties that require multiple quantifier alternations can be difficult, since these cannot be verified using self-composition, leading to a substantial decrease in verification performance. However, future work should focus on how to support other hyperproperties in Alloy and alternatives to self-composition.

Another possible future work is to merge this work with HAROS to duly support the ROS2 environment while making use of the various analysis techniques that HAROS supports. In particular, HAROS feedback interface is capable of providing more information than the svROS Visualizer. However, this requires a

substantial change in the DSL, which currently only allows a very loose specification of node behaviour.

The main objectives of this dissertation were met by offering a practical method for the automatic verification of OD. Despite the reasonable performance results in the *TurtleSim* case-study, exploring alternative Alloy idioms for the formalisation of OD might be helpful in improving the verification efficiency, so that the technique can be applied to real industrial cases.

Bibliography

[1] Alaa Alaerjan, Dae-Kyoo Kim, and Dhrgam Al Kafaf. Modeling Functional Behaviors of DDS. IEEE, 2017.

[2] Thomas A. Alspaugh. The Alloy Quick Reference, 2010. URL https://www.ics.uci.edu/~alspaugh/cls/shr/alloy.html.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[4] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. *Mathematical Structures in Computer Science*, 2011.

[5] Kai Beckman and Jonas Reininger. Adaptation of the DDS Security Standard for Resource-Constrained Sensor Networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, 2018.

[6] Dirk Beyer and Thomas Lemberger. Software Verification: Testing vs. Model checking. In *Haifa Verification Conference*. Springer, 2017.

[7] Jonathan Boren and Steve Cousins. Exponential growth of ROS. IEEE Robotics & Automation Magazine, 2011.

[8] Benjamin Breiling, Bernhard Dieber, and Peter Schartner. Secure Communication for the Robot Operating System. In *2017 annual IEEE International Systems Conference (SysCon)*. IEEE, 2017.

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

[10] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. Formal Software Design with Alloy 6, 2021. URL https://haslab.github.io/formal-software-design/.

[11] Gianluca Caiazza, Ruffin White, and Agostino Cortesi. Enhancing security in ROS. In *Advanced Computing and Systems for Security*. Springer, 2019.

[12] 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)*, 2020.

[13] 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.

[14] Edmund M Clarke. Model Checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1997.

[15] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model Checking and the state explosion problem. In *LASER Summer School on Software Engineering*. Springer, 2011.

[16] Lori A Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, 1976.

[17] Michael R Clarkson and Fred B Schneider. Hyperproperties. IOS Press, 2010.

[18] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*. Springer, 2014.

[19] Agostino Cortesi, Pietro Ferrara, and Nabendu Chaki. Static analysis techniques for robotics software verification. In *IEEE ISR 2013*. IEEE, 2013.

[20] Steve Cousins. Welcome to ROS Topics. IEEE Robotics & Automation Magazine, 2010.

[21] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the Internet for ROS: A View of Security in Robotics Research. In *2019 International Conference on Robotics and Automation (ICRA)*, 2019.

[22] Ruisheng Diao, Vijay Vittal, and Naim Logic. Design of a real-time security assessment tool for situational awareness enhancement in modern power systems. IEEE, 2009.

[23] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ROS-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[24] Bernhard Dieber, Benjamin Breiling, Sebastian Taurer, Severin Kacianka, Stefan Rass, and Peter Schartner. Security for the Robot Operating System. volume 98. Elsevier, 2017.

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

[26] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot Operating System 2: The need for a holistic security approach to robotic architectures. SAGE Publications Sage UK: London, England, 2018.

[27] Kyle Fazzari. ROS 2 DDS-Security Integration, 2020. URL https://design.ros2.org/articles/ros2_dds_security.html.

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

[29] Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with model checkers: A survey. Wiley Online Library, 2009.

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

[31] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982.

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

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

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

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

[36] Daniel Jackson. Alloy: A language and tool for exploring software designs. 2019.

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

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

[39] Leslie Lamport. Proving the correctness of multiprocess programs. IEEE, 1977.

[40] Leslie Lamport and Fred B Schneider. Verifying Hyperproperties With TLA. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021.

[41] 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)*. IEEE, 2018.

[42] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. 2016.

[43] 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*, 2018.

[44] Gerald A Marin. Network security basics. IEEE security & privacy, 2005.

[45] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *Proceedings of the 13th International Conference on Embedded Software*, 2016.

[46] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascarenas. A preliminary cyber-physical security assessment of the Robot Operating System. In *Unmanned Systems Technology XV*. International Society for Optics and Photonics, 2013.

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

[48] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking. In *International Static Analysis Symposium*. Springer, 1999.

[49] Object Management Group (OMG). Data Distribution Service (DDS).

[50] Object Management Group (OMG). DDS Security, 2018.

[51] Gerardo Pardo-Castellote, Bert Farabaugh, and Rick Warren. An introduction to DDS and data-centric communications. 2005.

[52] 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*. Kobe, Japan, 2009.

[53] Open Robotics. The Robot Operating System. URL https://www.ros.org/.

[54] Open Robotics. ROS 2 Documentation, 2021. URL https://docs.ros.org/en/foxy/index.html.

[55] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. IEEE, 2003.

[56] 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)*. IEEE, 2016.

[57] 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*, 2018.

[58] 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)*. IEEE, 2019.

[59] André Santos, Alcino Cunha, and Nuno Macedo. The High-Assurance ROS Framework. 2021.

[60] George Stavrinos. Robot Operating System Client Libraries, 2020.

[61] 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*, 2019.

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

[63] Thomas Wahl. Fairness and liveness.

[64] Yu Wang, Siddhartha Nalluri, Borzoo Bonakdarpour, and Miroslav Pajic. Statistical model checking for hyperproperties. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.

[65] Ruffin White and Mikael Arguedas. ROS 2 Security Enclaves, 2020. URL https://design.ros2.org/articles/ros2_security_enclaves.html.

[66] Ruffin White and Kyle Fazzari. ROS 2 Access Control Policies, 2021. URL https://design.ros2.org/articles/ros2_access_control_policies.html.

[67] Ruffin White, Dr Christensen, I Henrik, Dr Quigley, et al. SROS: Securing ROS over the wire, in the graph, and through the kernel. 2016.

[68] 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)*. IEEE, 2018.

[69] 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)*. IEEE, 2019.

[70] 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)*. IEEE, 2017.

[71] Steve Zdancewic and Andrew C Myers. Observational Determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* IEEE, 2003.

Appendix A
Alloy Model Translation

```
module TurtleSim

/* === SIGNATURES === */
abstract sig Node {
  subscribes, advertises : set Topic
}
abstract sig Not_Numeric {}
sig Message = Not_Numeric + Int {}
abstract sig Topic {}

/* === ROS ENTITIES : Nodes Topics Variables === */
one sig node_multiplexer extends Node {} {
  advertises = topic_move_turtle + topic_alarm
  subscribes = topic_safe + topic_move
}
one sig node_safety extends Node {} {
  advertises = topic_safe
  subscribes = topic_pose_log
}
one sig node_turtlesim extends Node {} {
  advertises = topic_pose_log
  subscribes = topic_move_turtle
}
one sig node_random extends Node {} {
  advertises = topic_move
  no subscribes
}
one sig node_light extends Node {} {
  no advertises
  subscribes = topic_alarm
}
one sig topic_move_turtle, topic_alarm, topic_move, topic_safe,
    topic_pose_log extends Topic {}
one sig True, False extends Not_Numeric {}
```

Figure 35: *TurtleSim* model in Alloy, namely "ros-concrete.als": Excerpt showing the ROS2 meta-model signatures.

```
pred multiplexer_low [t : Trace] {
  ((some t.inbox[topic_move]) and (no t.inbox[topic_safe]))
  let m = first[t.inbox[topic_move]] {
    ((t.inbox'[topic_move_turtle] = add[t.inbox[topic_move_turtle], m]))
    (( m = -2 or m = 2 ))
        implies { ((t.inbox'[topic_alarm] = add[t.inbox[topic_alarm], True
            ])) }
        else { ((t.inbox'[topic_alarm] = add[t.inbox[topic_alarm], False]))
             }
  }
  t.inbox'[topic_move] = rest[t.inbox[topic_move]]
  // Frame Conditions:
  all c : Topic - topic_move - topic_move_turtle - topic_alarm |
      t.inbox'[c] = t.inbox[c]
  t.position' = t.position
}
pred multiplexer_high [t : Trace] {
  ((some t.inbox[topic_safe]))
  let m = first[t.inbox[topic_safe]] {
    ((t.inbox'[topic_move_turtle] = add[t.inbox[topic_move_turtle], m]))
    (( m = -2 or m = 2 ))
        implies { ((t.inbox'[topic_alarm] = add[t.inbox[topic_alarm], True
            ])) }
        else { ((t.inbox'[topic_alarm] = add[t.inbox[topic_alarm], False]))
             }
  }
  t.inbox'[topic_safe] = rest[t.inbox[topic_safe]]
  // Frame Conditions:
  all c : Topic - topic_safe - topic_move_turtle - topic_alarm |
      t.inbox'[c] = t.inbox[c]
  t.position' = t.position
}
pred multiplexer [t : Trace] {
  ((no t.inbox[topic_move_turtle]) and (no t.inbox[topic_pose_log]))
  // Sub-Predicates:
  multiplexer_high[t] or multiplexer_low[t]
  // Frame Conditions:
}
```

Figure 36: *TurtleSim* model in Alloy, namely "ros-concrete.als": Excerpt showing behaviour of *Multiplexer*, translated into its predicate, with also its sub-predicates.

```
pred safety [t : Trace] {
  ((some t.inbox[topic_pose_log]))
  let m = first[t.inbox[topic_pose_log]] {
    (( m = -2 ))
        implies { ((t.inbox'[topic_safe] = add[t.inbox[topic_safe], 1])) }
        else { (( m = 2 ))
              implies { ((t.inbox'[topic_safe] = add[t.inbox[topic_safe], -
                 1])) }
              else    { ((t.inbox'[topic_safe] = t.inbox[topic_safe])) }
        }
  }
  t.inbox'[topic_pose_log] = rest[t.inbox[topic_pose_log]]
  // Frame Conditions:
  all c : Topic - topic_pose_log - topic_safe |
      t.inbox'[c] = t.inbox[c]
  t.position' = t.position
}
pred turtlesim [t : Trace] {
  ((some t.inbox[topic_move_turtle]))
  let m = first[t.inbox[topic_move_turtle]] {
    ((t.position' = plus[t.position, m]))
  }
  t.inbox'[topic_move_turtle] = rest[t.inbox[topic_move_turtle]]
  t.inbox'[topic_pose_log] = add[t.inbox[topic_pose_log], t.position']
  // Frame Conditions:
  all c : Topic - topic_move_turtle - topic_pose_log |
      t.inbox'[c] = t.inbox[c]
}
pred random [t : Trace] {
  some m : Message | (( m >= -2 ) and ( m =< 2 )) implies t.inbox'[
      topic_move] = add[t.inbox[topic_move], m] else t.inbox'[topic_move] =
      t.inbox[topic_move]
  // Frame Conditions:
  all c : Topic - topic_move |
      t.inbox'[c] = t.inbox[c]
  t.position' = t.position
}
pred light [t : Trace] {
  t.inbox'[topic_alarm] = rest[t.inbox[topic_alarm]]
  // Frame Conditions:
  all c : Topic - topic_alarm | t.inbox'[c] = t.inbox[c]
  t.position' = t.position
}
```

Figure 37: *TurtleSim* model in Alloy, namely "ros-concrete.als": Excerpt showing the rest of the node behaviour, translated into its predicate.

```
/* === SELF-COMPOSITION === */
abstract sig Trace {
    var inbox: Topic -> (seq Message),
    var position: one Var_Position
} one sig T1, T2 extends Trace {}

sig Var_Position in Int {}
-- Initial Configuration
fact system_behaviour {
    always (stutter[T1] or system[T1])
    always (stutter[T2] or system[T2])
}
pred publish[t: Trace, topic: Topic, m : Message] {
    not (t.inbox[topic].lastIdx = max[seq/Int]) // Only publish if inbox
        isnt full
    t.inbox'[topic] = add[t.inbox[topic], m]
}
-- Trace Execution
pred stutter [t : Trace] {
    t.inbox' = t.inbox
    t.position' = t.position
}
pred system [t : Trace] {
    // System executions.
    multiplexer[t] or safety[t] or turtlesim[t] or random[t] or light[t]
}
```

Figure 38: *TurtleSim* model in Alloy, namely "ros-concrete.als": Excerpt showing how self-composition is inferred and the initial configuration with regard to the latter. The only variable of the system is also captured in this excerpt: the position of the turtle which is identified as the Var_Position .

```
/* === OBSERVABLE DETERMINISM === */
fact public_state_equivalence {
    no inbox
}
fact initial_assumptions {
    T1.position = 0 and T2.position = 0
}
fact public_event_synchronization {
    always ( all m : Message |
             ( publish[T1, topic_move, m] )
             iff
             ( publish[T2, topic_move, m] )
    )
    always (
             ( some m0 : Message | publish[T1, topic_alarm, m0] )
             iff
             ( some m1 : Message | publish[T2, topic_alarm, m1] )
    )
}
check {
    always ( all m0, m1 : Message |
                    publish[T1, topic_alarm, m0] and publish[T2,
                       topic_alarm, m1]
                    implies m0 = m1
    )
} for 4 but 5 seq, 1..15 steps
```

Figure 39: *TurtleSim* model in Alloy, namely "ros-concrete.als": Excerpt showing how Observational Determinism is verified. Public-Event Synchronisation is applied to topics *move* and *alarm*. The verification of OD is exclusively applied to *alarm*, as this denotes a public observation. In this verification the inboxes scope (5 seq) and the steps scope (1..15 steps) is provided.

Appendix B
svROS Visualizer: Counterexamples

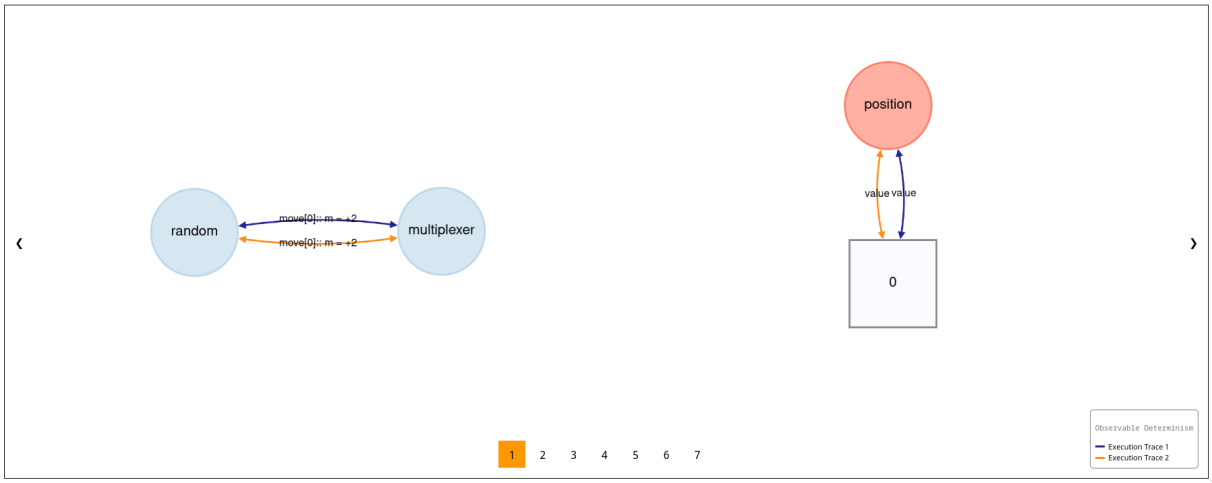


Figure 40: *TurtleSim*'s counterexample: *Random* issues a velocity message of **+2** to *Multiplexer*.

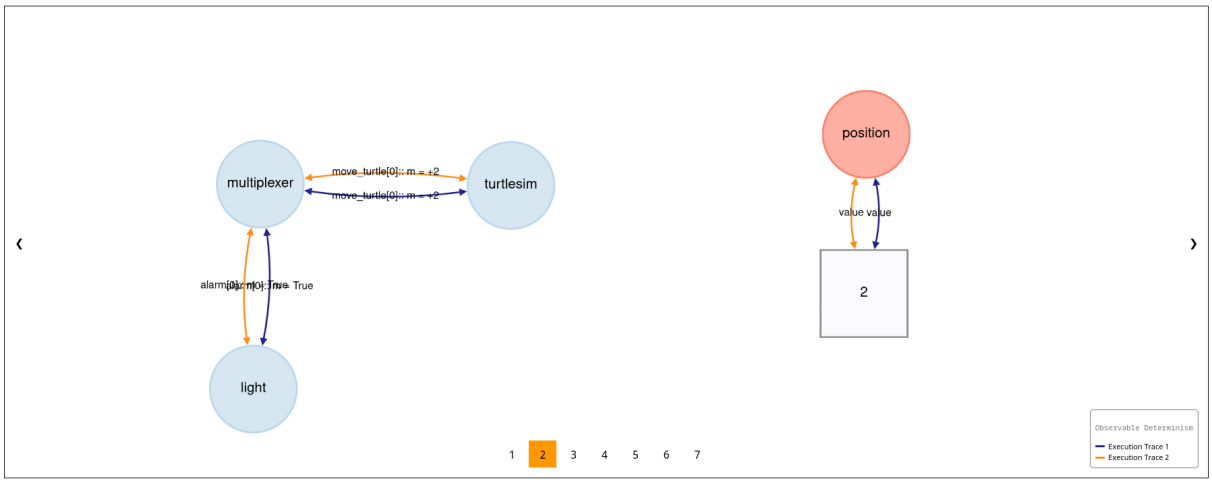


Figure 41: *TurtleSim*'s counterexample: *Multiplexer* forwards these messages to *Turtlesim*, who updates the turtle position to **2**. Simultaneously, *Multiplexer* fires a ringing alarm (True sent over the topic *alarm*).

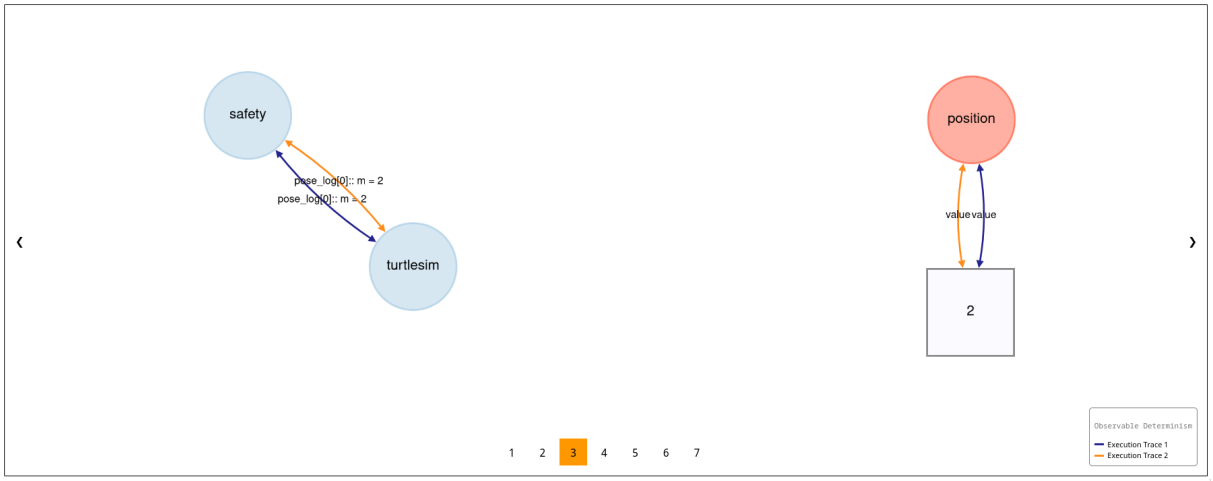


Figure 42: *TurtleSim*'s counterexample: *Turtlesim* forwards a message to *Safety* with the new position of the turtle (**2**).

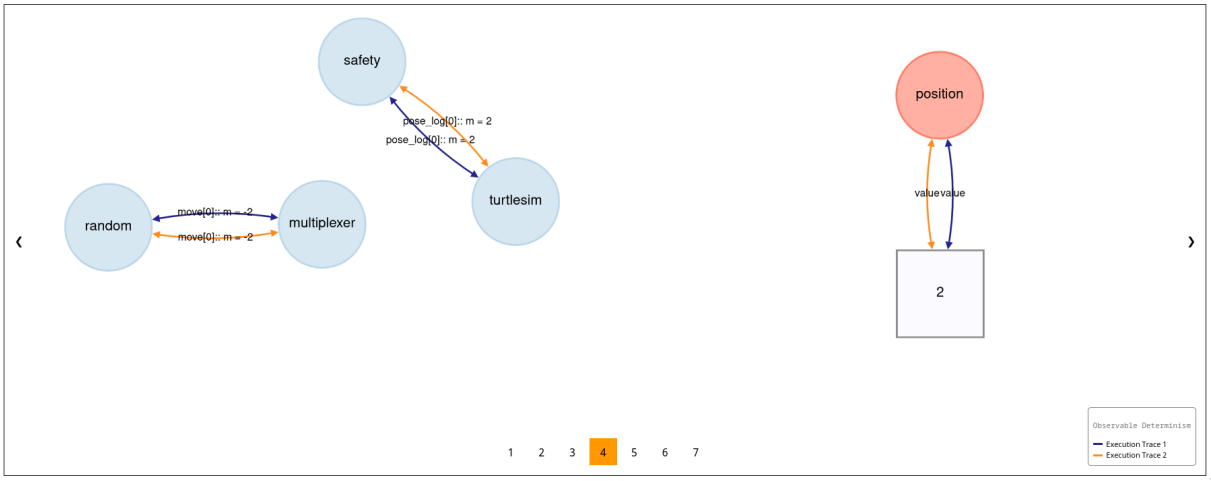


Figure 43: *TurtleSim*'s counterexample: *Random* issues another velocity message of **-2**.

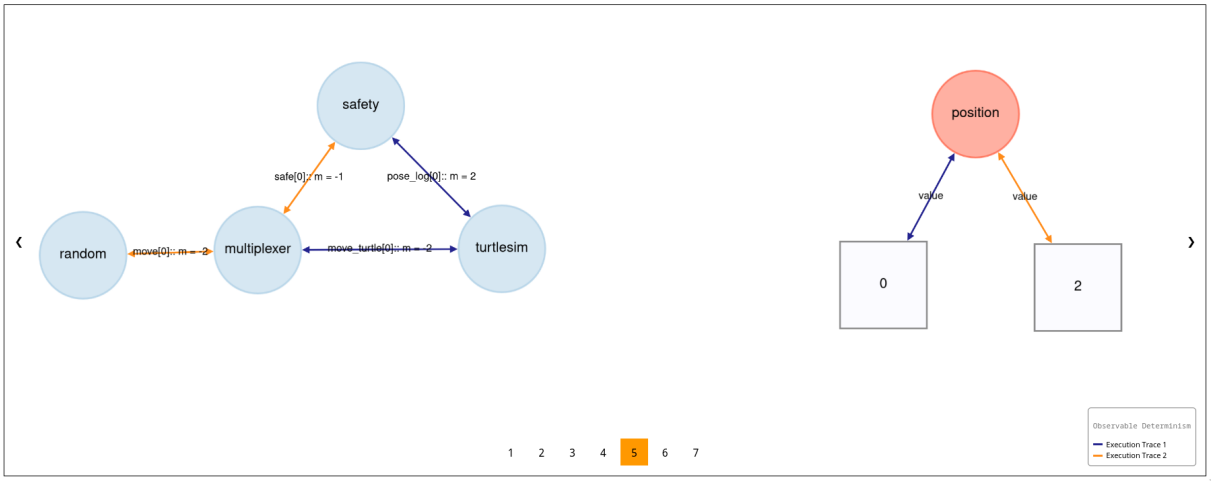


Figure 44: *TurtleSim*'s counterexample: In Trace 1, *Multiplexer* forwards the message arrived from *Random* to *Turtlesim*, who updates the position to **0**. In Trace 2, *Safety* reacts first and sends a correcting message of **-1** to remove the turtle from the border.

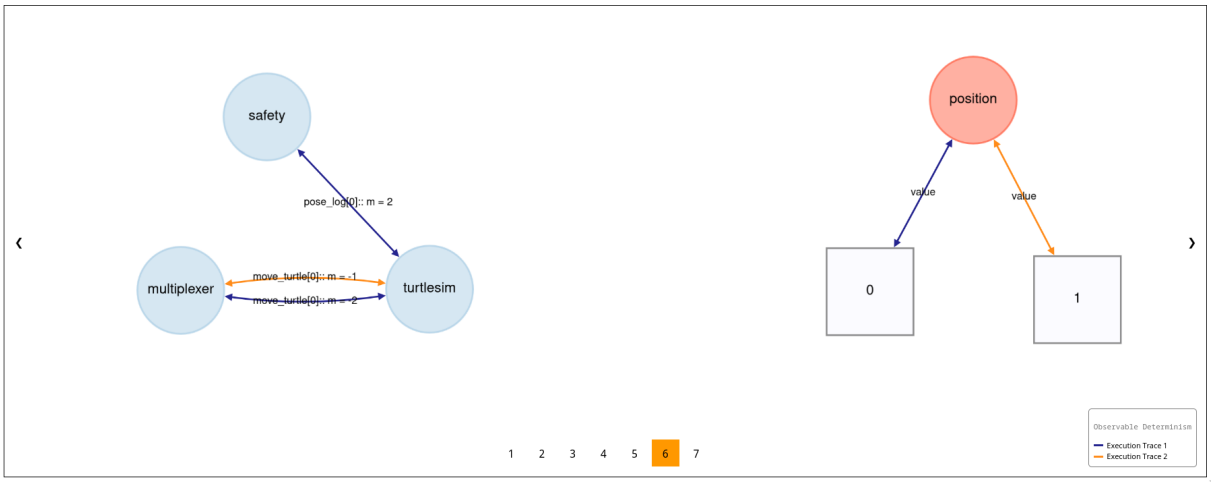


Figure 45: *TurtleSim*'s counterexample: In Trace 1, it stutters. In Trace 2, *Multiplexer* forwards the message arrived from *Safety*, where *Turtlesim* updates the position from **2** to **1** (removes from the border).

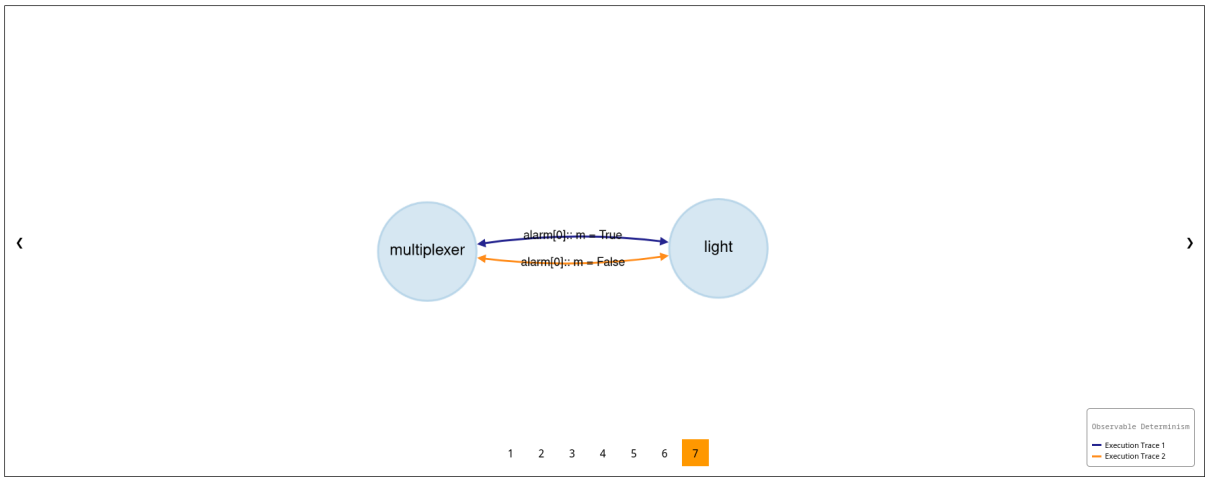


Figure 46: *TurtleSim*'s counterexample: In Trace 1, *Multiplexer* forwarded a high velocity message that arrived from *Random*, which causes the sending of a ringing message (True sent over the topic *alarm*). Whereas in Trace 2, *Multiplexer* forwarded the message that arrived from *Safety*, which causes the sending of a not-ringing message (False sent over the topic *alarm*).