arXiv:2208.05507v4 [cs.LO] 31 Jul 2026

A Compositional Approach to Verifying Modular Robotic Systems

Matt Luckcuck [*1], Marie Farrell [2], Angelo Ferrando [3], Rafael C. Cardoso [4], Louise A. Dennis [2], and Michael Fisher [2]

[1]School of Computer Science, University of Nottingham, United Kingdom
[2]Department of Computer Science, University of Manchester, United Kingdom
[3]Department of Physical, Computer and Mathematical Sciences, University of Modena and Reggio Emilia, Italy
[4]Department of Computing Science, University of Aberdeen, United Kingdom

3rd August 2026

Abstract

Robotic systems used in safety-critical scenarios often rely on modular software architectures, and increasingly include autonomous components. Verifying that these modular robotic systems behave as expected requires approaches that can cope with, and preferably take advantage of, this inherent modularity. This paper describes a compositional approach to specifying the nodes in robotic systems built using the Robot Operating System (ROS), where each node is specified using First-Order Logic (FOL) assume-guarantee contracts that link the specification to the ROS implementation. We introduce inference rules that facilitate the composition of these node-level contracts to derive system-level properties. We also present a novel Domain-Specific Language, the ROS Contract Language (RCL), which captures a node's FOL specification and links this contract to its implementation. RCL contracts can be automatically translated, by our tool Vanda, into executable monitors; which we use to verify the contracts at runtime. We illustrate our approach through the specification and verification of an autonomous rover engaged in the remote inspection of a nuclear site, and finish with smaller examples that illustrate other useful features of our framework.

1 Introduction

Robotic systems are increasingly deployed in industrial, often safety-critical, scenarios such as monitoring offshore structures [1], nuclear inspection and decommissioning [2, 3], and space exploration [4, 5]. Engineering the software to control a robotic system is a complex task, often supported by modular software frameworks, such as the Robot Operating System (ROS) [6] or G$^{\text{en}}$oM [7, 8]. It is crucial to ensure that the software controlling a robot behaves correctly, particularly as modern robotic systems become more autonomous, more complex, and are used in dynamic environments that they share with

*This work began in 2018 while all authors were employed by the University of Liverpool, UK. The work was supported by UK Research and Innovation, and EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE), EP/R026173 (ORCA), and EP/R026084 (RAIN); and through grant EP/V026801 (TAS Verifiability Node). Cardoso's and Fisher's work was supported by the Royal Academy of Engineering under the Chairs in Emerging Technologies scheme. some of Luckcuck and Farrell's work was performed while employed by Maynooth University, Ireland; and The University of Manchester, UK. Some of Ferrando's work was performed while employed by the University of Genova, Italy.

To ensure open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

The formal models, data, and source code supporting the findings reported in this paper are openly available from the Zenodo repository at: `https://doi.org/10.5281/zenodo.6941344`.

humans. The generality and flexibility of robotic software frameworks also means that guaranteeing their correct behaviour is challenging [9].

The state-of-the-art for verification of autonomous and robotic systems includes a variety of formal methods that can be used for specification and verification [10] – non-formal methods such as field tests and simulation-based testing are also common. Formal methods that we see being used include: model-checking [11], which exhaustively explores the state space to establish that a property holds; runtime verification [12], which monitors system behaviour at runtime; and theorem-provers [13], demonstrating by mathematical proof that the system behaves correctly. Different components of a robotic system may be better suited to different verification techniques, but linking the outputs of multiple techniques remains a challenge. Previously, we have argued that robotics is a domain in which *integrating* (formal and non-formal) verification methods is both a necessity to be dealt with and an opportunity to be grasped [14].

Assume-Guarantee reasoning [15] (or the specification of *pre-* and *post*-conditions) is a well established compositional verification technique. A pair of pre- and post-conditions form a *contract* [16]. Specifying a system using Assume-Guarantee or *pre-* and *post*-conditions enables it to be decomposed into modules, so that each module can be verified separately against its associated contract. However, care must be taken when specifying the contracts, because they still require validation against the actual requirements of the system.

This paper presents our approach to verifying robotic systems that are developed using ROS. This particular robotic software framework was chosen because of its prevalence in the literature. A ROS system is composed of nodes that communicate using message passing via buffered communication channels. The nodes coordinate to control the robot's overall behaviour. Typically, each node will be specialised to perform a different function, with different nodes (or collections of nodes) often requiring distinct verification techniques. For example, machine learning components will likely be verified via testing, whereas a planner might be mathematically modelled and reasoned about using formal verification. Our approach uses the encapsulation provided by ROS nodes to provide compositionality. Fundamentally, we address the research question:

Can we use a compositional and heterogeneous approach to verify ROS-based systems?

Our approach begins by (manually) abstracting the graph of nodes in the ROS program (generated by ROS Graph or similar) into a more manageable model of the system, a model that focusses on the system's most critical components. Although some nodes maintain a one-to-one correspondence with the ROS software, abstraction can involve dropping some nodes from the system model or combining related nodes into, what we call, a *compound node*.

Once the ROS software is abstracted to a manageable system model, we specify each of the nodes in this model using Assume-Guarantee contracts, written in our contract specification language that is based on First-Order Logic (FOL). FOL was chosen for contract specifications because it is both expressive and widely understood, flattening our approach's learning curve.

We also provide a calculus that can be used to combine the module contracts and to derive system-level properties that correspond to the system's requirements. The calculus uses temporal operators from First-Order Temporal Logic (FOTL) [17] to represent the connections between the contracts.

Once the contracts are verified against the system's requirements, we can take two further steps in parallel. The contracts are used to *guide* the verification of each node, using heterogeneous (formal or non-formal) verification approaches chosen to suit each node. Providing a formal link between FOL and every verification approach is not in scope for this paper. Using the contracts as a guide enables the use of formal and informal links between the contracts and verification. We leave the verifiers to choose the most suitable verification method for each node, because they are best placed to make this choice. Meanwhile, our tool, Vanda, automatically synthesises runtime monitors from the contracts. The overall approach builds on two pieces of previous work:

- the initial presentation of the contract calculus in [18], which we significantly update in this paper; and,
- our application of heterogeneous verification approaches to a simulation of the Mars Curiosity rover [19], which we also update with additional steps.

Our work provides four contributions:

1. A compositional approach to specifying the pre- and post-conditions of robotic systems constructed using ROS, which is supported by;

2. a Domain Specific Language (DSL), called the ROS Contract Language (RCL), that links a contract to the ROS implementation;
3. a calculus containing inference rules for combining the contracts so that we can derive system-level (safety and mission) properties; and,
4. a tool-chain that synthesises runtime monitors from the system's RCL contracts.

Our approach enables the introduction of a formal specification to an existing ROS system. As such, we use the intended behaviour of the nodes in a system as the starting point for specifying its contracts, and combine them using the calculus. Our DSL and prototype parsing and monitor-generation tool (Vanda) support users in writing grammatically correct contracts.

In summary, the contracts: are structured using RCL; are reasoned about using our calculus; guide the heterogeneous verification of the nodes; and are then used to generate runtime monitors that provide a safety net, ensuring that the contracts have been verified correctly and the system is obeying its requirements. We validate our approach by applying it to a rover robot performing a remote inspection task inside a nuclear storage facility (§4). This example system was developed independently of our work, and we use our approach to introduce contracts and formal verification. We also describe the specification and verification of individual nodes, showing how our approach can guide verification using a variety of different formalisms.

The remainder of this paper is structured as follows. In §2, we discuss related work. §3 describes our compositional approach to verifying modular robotic systems using FOL contracts, including: the calculus for combining node specifications, a description of how RCL supports contract specification, and the automatic synthesis of runtime monitors from RCL. §4 presents the specification and verification of our Case Study, a remote inspection rover. In §5 we discuss some interesting characteristics of our framework, such as the use of a system's modularity, how our approach fits into the robotic software development process and, in §5.2, we describe how our approach can be applied to non-ROS systems. Finally, §6 concludes the paper and presents avenues for future work.

2 Related Work

This section discusses approaches in the literature that are related to our work. We have grouped these into Compositional Verification and Reliable Software Engineering (§2.1), which covers work using assume/guarantee contracts; and Robotics (§2.2), which covers work relating to specifying robotic systems specifically. Some of the cited work could fit into both of these categories, but we have added them to the most relevant category based on their main contributions and publication venue.

2.1 Compositional Verification and Reliable Software Engineering

Our approach encourages the development of systems as a composition of sub-systems, as does the work on the Pacti [20] tool for assume-guarantee contracts. Pacti supports polyhedral constraints in which the terms are expressed as linear inequalities with real coefficients. Pacti also supports several contract operations (for example refinement and composition) to reason about the relationship between contracts. One key feature of Pacti is that the contract operations are agnostic of the specific algebra used, though the work in [20] only implements polyhedral constraints so far, with other formalisms "such as LTL and nonlinear constraints" suggested as future work. In contrast to the language used by Pacti, we use FOL which is more expressive and actually capable of expressing Pacti contracts. However, Pacti's contract operations are more developed than ours, as it includes refinement between contracts and a *quotient* operator that can discover the specification of a missing subsystem that will combine with the existing specification to meet a top-level system specification. Our work focusses on specifying contracts that guide the verification of the system and enable the automatic synthesis of Runtime Verification (RV) monitors, both of which are absent from the work on Pacti.

Compositional verification is applied to the SIENA event-notification middleware [21] where a global system property is decomposed into local properties that only hold on sub-parts of the system. They use compositional model-checking and the system models (labelled transition systems) are translated to Promela, using SPIN for verification. The various properties are related using simulation, a notion that in [21] seems similar to formal refinement. Our work provides a broader approach; we are not restricted to model-checking and we enable heterogeneous verification. That said, their way of relating global and sub-properties is interesting and will likely inspire future directions for our work.

Compositional Assume-Guarantee reasoning has been used to define contracts for system modules [22], with rule-defined contract composition. Their rules share the same aim as our work and are extended by [23], using a variant of Signal Temporal Logic (STL) to describe behaviour and contracts. They used the rules in [22] to produce whole system assumptions and guarantees. They target closed-loop control systems, whereas our approach targets robotic systems that are written in general-purpose programming languages and leverages heterogeneous specification. Related compositional approaches include OCRA [24] and AGREE [25], though neither explicitly incorporates heterogeneous verification.

CoCoSpec [26] is a language that provides assume/guarantee contracts for reactive systems. CoCoSpec extends the Lustre specification language and uses the Kind2 model-checker for compositional verification [27]. This approach is specialised for synchronous communications, which differ from the event-based communications that we target, and their contract semantics is more restrictive than ours. Specifically, CoCoSpec with Kind2 uses logical implication requiring the user to show that the lower-level node specifications imply a predefined top-level node specification; whereas we derive the top-level contract from the combination of the lower-level contracts. Further, it is not clear how their support for compositional verification can be extended to support heterogeneous components such as those in our example.

In previous work by some of the authors of this paper, a combination of NASA's Formal Requirements Elicitation Tool (FRET), CoCoSpec, and Event-B were used to verify an inspection rover example system [28]. The work used a Simulink model of the rover's architecture to define individual components. The combination of techniques was needed to adequately verify the rover system. For example, the CoCoSpec model could not be verified against the planner requirements for grids larger than 4×4 so Event-B's proof-based approach was used to verify these properties for any size of grid. However, doing so forced us to verify an abstracted planner model in place of the Simulink model, potentially sacrificing some accuracy for tractability. In contrast, our work focusses on systems developed in ROS and uses FOL contracts to reason about system-level properties.

We take inspiration from Broy's approach to systems engineering [29] which presents three kinds of artefacts: (1) system-level requirements, (2) functional system specification, and (3) logical subsystem architecture. These are represented as logical predicates in the form of assertions, with relationships defined between them that extend to assume/commitment contracts. The treatment of these contracts is purely logical, and we present a similar technique that, instead of assertions, uses Assume-Guarantee contracts and is specialised to the software engineering of robotic systems.

Ruchkin et al. describe the Integration Property Language (IPL) [30], which also uses FOL with temporal operations to specify *integration properties* (properties involving multiple models and formalisms) for heterogeneous models, specifically aimed at cyber-physical systems. They target *architectural views* (abstracted, behaviourless component models that are annotated with types and properties) which handle the difficult task of integrating heterogeneous models. IPL specifications can be verified using Satisfiability Modulo Theories (SMT) solvers and model checkers. Our approach avoids the difficulty of integrating the node-level models by combining the nodes, more abstract, FOL contracts and verifying the heterogeneous models of each node in isolation.

Publish-subscribe architectures, like ROS, are popular in many domains. Baresi, et al. present the Loupe model-checker for publish-subscribe architectures [31]. They essentially embed the communications infrastructure within the verification checker to reduce the state space for verification. Such approaches are certainly relevant for our work, but, they do not explicitly focus on ROS or support a compositional approach to verification of individual system modules.

2.2 Robotics

Many approaches for building safe robotic systems focus on ROS, which is a well established middleware that supports interoperability and modularity in the development of robotic software. A safety-critical working group[1] for ROS2 has been developing tools, libraries, and documentation to support the safe engineering of safety-critical ROS systems. For example, they provide a *contracts* package[2], where a contract is a combination of pre-/post-conditions and assertions over the implementation of C++ functions. The advantage of this approach is that the contracts work directly in the implementation's source code. However, limiting the library to C++ is a disadvantage because ROS nodes may also

[1] `https://github.com/ros-safety` Accessed: 03/11/2023

[2] `https://github.com/ros-safety/contracts_lite` Accessed: 03/17/2023

be written in Python, Java, etc. In comparison, our approach is more general and not limited to the verification of individual functions, but also allows the verification of the system as a whole through the use of our inference rules. Further, their ad-hoc contract language is less expressive than FOL. An interesting line of future work may include updating VANDA, our prototype tool, to produce contracts compatible with the ROS2 contracts package for applications fully implemented in C++.

A similar approach is shown in [32] for G$^{\text{en}}$oM instead of ROS, which uses the Behaviour-Interaction-Priority (BIP) framework for incremental composition of heterogeneous components. They offer synthesis of functional-level controllers by synchronising dependencies between controllers. They verify safety properties and detect deadlock conditions using model checking and "observers" (runtime monitors). Another compositional approach in [33] uses Assume-Guarantee contracts to decompose the control software of multi-robot systems and targets ROS. The individual components or robots are decomposed into sub-problems and then recomposed using contracts to provide system-level validation. Finally, the resulting synthesised controller is integrated into ROS.

Drona [34] is a toolchain for programming safety-critical robots, with support for ROS. Their DSL, called P, is based on state machines. It offers compositional Assume-Guarantee testing and a runtime assurance system to check that the assumptions made at design-time hold at runtime. In contrast, our DSL is used purely to specify contracts for verification, we do not directly interfere with the system's implementation. Additionally, our runtime monitors are automatically synthesised from the contracts, while theirs require additional specification.

The Declarative Robot Safety (DeRoS) [35] is a DSL with a declarative syntax for specifying safety-related constraints in ROS, which lowers the barrier to using this approach – similarly to our use of a textual version of FOL. In DeRoS, each contract is a refinement of the system-level contract; it is not clear if they use a similar technique to our inference rules to ensure that the contracts are composed correctly at design time. Unfortunately, DeRoS is not publicly available, or it could have been an alternative way to synthesise our monitors.

Another DSL, PROMISE [36], is designed to describe mission specifications for multi-robot systems. PROMISE has been integrated in Eclipse as a plugin to provide a graphical interface for users which allows the automatic generation of behaviour from the mission specification, sending missions to the robots, and runtime management of missions.

Other related work provides automatic static verification of system-wide properties for message-passing in ROS applications [37]. The specification of the safety properties is written in a DSL. This is then translated into FOTL to be used in Electrum [38] which provides an automatic *Analyser*. Their approach is embedded in HAROS [39], a framework for quality assessment of ROS software that offers a visualisation interface for safety issues. Instead, our work focuses on compositional verification of nodes/modules in ROS, which are supported by inference rules to automatically generate system-wide properties. Further, we are not limited to static verification, our approach also provides runtime verification by synthesising monitors from the contracts.

SOTER [40] is a programming framework to support the development of robotic systems by capturing runtime safety assurance principles. Their high-level DSL can be used to implement reactive systems, make use of systematic testing techniques, and support runtime assurances. The case study and experiments presented use ROS. Our approach differs in that we are not limited to runtime assurances, we also encourage a variety of offline verification techniques.

RoboSC is a DSL, and accompanying tool, that enables the specification of ROS nodes as Finite-State Automata (FSA) and then synthesises supervisory controller nodes [41]. The events in the FSA are the ROS communication topics that trigger state changes in the automaton. The supervisory controller nodes aim to enforce a node's (user-described) requirements. RoboSC always adds the overhead of communication to the ROS middleware, which increases the average time of a controller cycle by approx. 957% (adding an average of 33.5 microseconds). Our approach can be used to enforce requirements, or to simply observe and log deviations from the specification; the latter option avoids some of the overheads of the additional nodes needed for RV.

RoboChart [42] is a DSL based on the Unified Modelling Language (UML), which supports verification and automated reasoning of robotic systems using model checking and theorem proving. Its notation is based on state-machines, with a restricted set of constructs. Integrated Development Environment (IDE) support is available through an Eclipse plugin called RoboTool that automatically generates C++ code for state machines and controllers, but does not yet offer automatic code deployment in ROS.

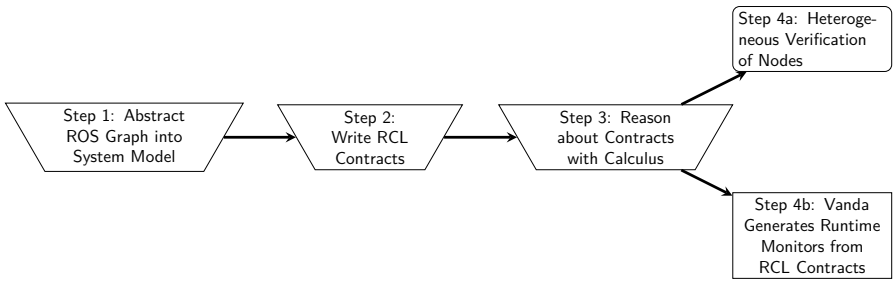


Figure 1: A flowchart of the steps in our approach. The trapezium-shaped steps are manual processes, the rectangle with rounded corners (Step 4a) is a process that may be manual or automatic (depending on the verification tools available), and the rectangle with right-angled corners (Step 4b) is an automatic process. Note that Steps 4a and 4b can happen in parallel. If needed, previous steps may be revisited.

As robotic systems become more complex, supporting heterogeneous verification becomes even more important [14]. Crucially, the modular structure of ROS systems facilitates the use of heterogeneous verification methods. For example, recent work used various verification techniques on an autonomous space debris removal grasping system that was developed in ROS [43]. Another example is Antlab [44], a multi-robot task server for declarative multi-robot programming based on ROS. Our calculus could be used to derive system-level contracts for systems like these.

2.3 Summary

Many approaches partially address the challenge that we tackle, and there are various foundational approaches to compositional verification that are not restricted to any particular domain. We take inspiration from some of these, but devise a calculus that is specifically tailored for ROS systems. Other approaches are limited to specific tools and so neither support nor harness the power of incorporating a suite of heterogeneous verification approaches in the way that our approach does. To ensure traceability and consistency, we also provide a way of automatically generating runtime monitors so that we can support both static and dynamic verification for systems that operate in the real world.

3 Specifying Verifiable Robotic Systems

As mentioned in §1, our work enables the introduction of a formal specification to existing ROS programs. Recall that a ROS system is composed of *nodes*. Each node may subscribe to receive messages from, or publish messages to, a *topic* (a buffered communication channel). Each topic is described by the message type(s) that it can accept. ROS contains several built-in message types such as `string`, `bool`, and `int8`; and custom types can also be added.

Our verification approach begins with a manual analysis of the ROS program, to abstract it into a more manageable system model. We then specify a contract for each of the system's nodes in *typed* FOL, which was chosen to lower the barrier to learning to write contracts. The contracts are then combined and reasoned about using our calculus, which uses the $\bigcirc$ ("next") and $\Diamond$ ("eventually") temporal operators from FOTL. We use the contracts to guide the modelling and verification of individual nodes, while our tool, Vanda, automatically synthesises monitors from the contracts to provide a safety-net that checks that their guarantees are obeyed at runtime.

Our approach is split into the following five steps (shown in Fig. 1):

Step 1 (Manual): Abstract the ROS program into a more manageable system model, containing the nodes that are critical to the program's correct behaviour. Depending on the ROS program, this step may involve some form of abstraction, for example by combining multiple nodes into a *compound node*, or omitting nodes that are known to be reliable.

Step 2 (Manual): Write the RCL contract for each node. The contract contains the FOL *assume* and *guarantee* conditions, and describes the ROS topics that the node uses (its inputs and outputs). Our RCL tool, VANDA, parses the contracts, identifying where the format of the language has not been adhered to, and can synthesise runtime monitors (see Step 4b).

Step 3 (Manual): Use our calculus to reason about the combination of the node contracts. This generates a list of system-level properties that can be used to verify the system's requirements. The calculus also identifies malformed contracts, which we debug until the inference rules are valid. This includes checking/verifying that any simplifying assumptions made to successfully apply the proof rules are met.

Step 4a (Semi-Automatic): Verify the nodes, using a suite of heterogeneous verification approaches. This step uses the contracts to guide the verification, which is especially important where there is not a formal link between the verification approach and FOL.

Step 4b (Automatic): Automatically synthesise runtime monitors for nodes from their RCL contracts. These monitors verify the contracts' guarantees at runtime. The assumptions are accounted for during the compositional reasoning step, where the calculus verifies that a node's assumptions are satisfied by the guarantees of the node(s) that provide its inputs.

This paper focuses on ROS systems, but Steps 1 and 2 above could be applied to a system where the nodes/modules are classes or methods. We discuss how this might work in §5.2.

The steps are numbered sequentially, but they are *not* intended to be only followed linearly; Steps 4a and 4b can also be performed in parallel. Additionally, previous steps may be revisited, if needed. For example, as identified above, Step 3 could highlight a malformed contract that would be debugged and potentially rewritten, which revisits Step 2. Similarly, Step 4a could reveal that a specification is too restrictive to be verified (for example), which could trigger a re-write (revisiting Step 2) or the restrictive part of the specification could be left to be monitored in Step 4b. Step 1 can also be revisited if we later find that an abstraction in the system model is troublesome, for example we may need to split a compound node back into its constituent parts.

It is important to note the difference in purpose between Steps 3 and 4a/4b. In Step 3, the calculus is used to combine the contracts and reveal the system-level property (or properties) that is produced by the combination of their guarantees. The verification in Step 4a statically verifies that the nodes implement their contracts, which then implies the system-level property holds. And the monitors in Step 4b check the guarantees hold at runtime; if all the monitors do not conclude *false*[3], then the system-level property holds. If one monitor concludes *false*, then we know that the monitored guarantee has been violated. In many cases, this also identifies the software node responsible for not satisfying the contract. However, a guarantee violation may also be the consequence of an unmet assumption. In our approach, this risk is mitigated by the compositional reasoning in Step 3, since the calculus checks that the assumptions of connected nodes follow from the guarantees of the preceding nodes. Thus, for most internal software interactions, an assumption violation corresponds to a guarantee violation of another monitored node. Assumptions of environmental inputs require separate consideration, and explicit assumption monitors could provide further diagnostic information in such cases.

In the remainder of this section we describe: the process of abstracting a ROS system into a system model, §3.1; how our contracts are described and composed, §3.2; our approach to writing the contracts, §3.3; our calculus for combining the contracts, §3.4; how the contracts can guide heterogeneous verification, §3.5; and our runtime monitoring approach, §3.6.

[3]Because our runtime monitors are checking for violation of a property, they return *false* when no violation is detected.

3.1 System Model

To start writing the contracts we need a description of system's nodes. Because we are using ROS, we can make use of the *rqt_graph* library[4], which automatically generates a graph (called a ROS graph) for the system that contains all of its nodes and the communication links between them. ROS graphs can also display topics and actions (used to execute long-running tasks), but using only the nodes and communication links is enough for our purposes. If the graph is simple enough, then we can use it as the system model. However, the graph is often very large, with many nodes from well-tested libraries, so we might choose to abstract the ROS graph into a more manageable system model.

This section describes a heuristic approach to generating a more compact system model based on the ROS graph.

1. *Generate the ROS graph:* use the *rqt_graph* library to generate a graph of the system. This can be used "as is" (skip to Step 3 in this methodology) or can be further abstracted as detailed in the next steps.

2. *Remove nodes:* remove nodes that match the following conditions:

   - nodes from libraries that have been demonstrated to be reliable in most cases (e.g., the *move base* library[5] for path planning in ROS) through community experimentation and testing;
   - and nodes that are simple or have no impact on the nodes or properties that are being verified.

3. *Combine related nodes into compound nodes:* some nodes may be simple parts of a larger group or sub-system. These nodes can be merged into a *compound* node, making sure that it retains all the information needed for verification and that it matches the implementation of the original nodes.

4. *Add external nodes:* some nodes may be external to ROS, such as autonomous components (e.g., rational agents) and image processing (e.g., machine learning), and therefore do not appear on a ROS graph. These nodes are added to the system diagram, alongside a description of how they interact and communicate with the other nodes.

This heuristic approach indicates how a more tractable system model can be distilled from a complicated ROS graph. The most important aspect is that the abstracted system model must still resemble the implementation of the ROS nodes. We provide an example application of this heuristic in §4.1.

3.2 Background Concepts for the First-Order Logic Contracts

Our contracts use the standard definition of FOL with quantifiers ($\forall$, $\exists$) and logical connectives ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) over logical propositions including basic set theory [45]. For a given component/node, C, its contract comprises $\mathcal{A}_C(\overline{i_C})$ (assumption/pre-condition) and $\mathcal{G}_C(\overline{o_C})$ (guarantee/post-condition), where $\overline{i_C}$ and $\overline{o_C}$ are each a vector of variables representing the node's inputs/outputs, respectively. (Note that, when discussing one specific component, we often omit the C subscript.)

Complex robotic systems often produce and consume *streams* of data. Our approach to stream semantics is based on well-established work in the area of stream logic programming [46, 47]. In our approach, a stream is a list of data: $[e \mid \mathit{Tail}]$, where e denotes the first element in the list and *Tail* denotes the remaining elements. When receiving data, a component takes e from the stream, processes it, and recurses over *Tail*.

Each contract states that if a node consumes input data $\overline{i}$ from its input stream ($\mathsf{InStream}([\overline{i} \mid S])$) and $\mathcal{A}(\overline{i})$ holds (i.e., $\overline{i}$ satisfies the assumption/pre-condition for correct operation of the node) then eventually the node will place some data, $\overline{o}$ on its output stream that satisfies the node's guarantee. So if its current output stream was T, i.e. $\mathsf{OutStream}(T)$, before the execution of the node[6] functionality on $\overline{i}$, then afterwards the output stream will be $\mathsf{OutStream}([\overline{o} \mid T])$ and $\mathcal{G}(\overline{o})$ will hold. While our contract assumptions and guarantees are expressed using FOL, we represent the meaning of the contract

[4] *rqt_graph* library: `http://wiki.ros.org/rqt_graph` Accessed: 03/11/2023
[5] Move Base ROS Library: `http://wiki.ros.org/move_base` Accessed: 03/11/2023
[6] T is the sequence of outputs so far.

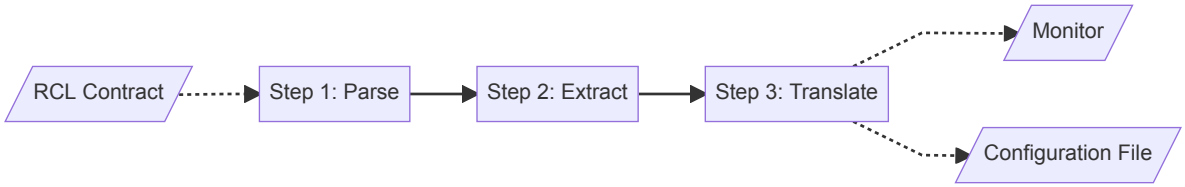


Figure 2: A flow chart showing VANDA's monitor synthesis workflow. The rectangles represent steps in the workflow, and the solid lines show the flow of control between the steps. The parallelograms represent input or output files, and the dotted lines show the file being input to or output from a step.

by a small extension using the "eventually" operator, '$\Diamond$' from Linear-time Temporal Logic (LTL) [48]. Thus, a contract guarantees:

$$(\mathsf{InStream}([\overline{i} \mid S]) \wedge \mathcal{A}(\overline{i}) \wedge \mathsf{OutStream}(T)) \Rightarrow \Diamond(\mathsf{InStream}(S) \wedge \mathcal{G}(\overline{o}) \wedge \mathsf{OutStream}([\overline{o} \mid T]))$$

for any streams S and T.

Using the temporal logic $\Diamond$ operator enables us to abstract away from internal computation/activity. Since the internal computation will never be instantaneous, nor do we have precise timing constraints, the execution of one component is described as *eventually* completing. Hence the use of the "sometime in the future" temporal operator '$\Diamond$'. Later in the development process, this very general temporal constraint might be refined to more precise real-time computational properties.

As in Stream Logic Programming, we will have rules to deal with end cases, such as when the remaining input list/vector is empty. Note that in such cases we might choose to terminate the processing, to suspend and wait until the list or vector is non-empty, to perform some exception handling, or undertake any other required computation. Note also that we make the simplifying assumption that each contract consumes one element from the input stream and generates (at most) one element for the output stream. In practice, this can be generalised to contracts consuming multiple input elements either by combining the input elements into a new (compound) element or by having several sub-contracts to handle the different input elements (as has been shown within Stream Logic Programming languages). However, for simplicity of description we keep our assumption that one input element is consumed at a time.

Also, a component can consume, or produce, multiple streams. For example, we might have a component consuming items from several streams to generate a combined output (on one stream). Whether we take an item from each stream simultaneously or just take an item from one of the streams (or any combination of these approaches) will depend on the component, and the component verification should account for this where appropriate. Note that we deliberately say nothing about the global behaviour of concurrent streams, for example whether one is generated more quickly than another, instead focusing on the first element on each relevant stream. These concurrency aspects might well be explored in future work but, for this initial investigation, we concentrate on the straightforward case for clarity of explanation.

Although we use FOL to specify contracts, we require a machine-readable syntax for capturing and generating monitors for contracts. For this, we introduce RCL in the next subsection.

3.3 Specifying Nodes in the ROS Contract Language

RCL captures a node's FOL contract; plus a description of the node's inputs and outputs, and the ROS topics to which they correspond. The contracts describe the behaviour of a node at a higher level of abstraction than the ROS implementation or a task/mission specification. The RCL contracts are agnostic of the communication type between ROS nodes (topics or services, for example) but the link in the RCL contracts between inputs/outputs and a corresponding ROS topic enables the synthesis of RV monitors. This work uses a version of our chosen RV framework (see §3.6) that only monitors topics, the stream abstraction we describe in §3.2 does not prevent an extension that can capture ROS services and actions. We discuss this possible extension, supported by a recent update to the RV framework, further in §3.6.

start :	contract_clause+
contract_clause :	node_clause | context_clause
context_clause :	“context” “{” (type_declaration | constant_declaration)+ “}”
type_declaration :	STRING “:” type_declaration_part “;"
constant_declaration :	STRING “=” type_declaration_part “;”
type_declaration_part :	? a function declaration, set, sequence, or tuple ?
node_clause :	“node” STRING “{” inputs outputs topic_list (assume)* (guarantee)+ “}”
inputs :	“inputs” “(” (io_var (“,” io_var)*)? “)”
outputs :	“outputs” “(” (io_var (“,” io_var)*)? “)”
io_var :	STRING “:” STRING
topic_list :	“topics” “(” (topic (“,” topic)*)? “)”
topic :	TYPE STRING (“matches” “(” topic_match_name “)”)?
topic_match_name :	io_pointer? STRING
io_pointer :	“in.” | “out.”
assume :	(“assume”) “(” formula “)”
guarantee :	(“guarantee”) “(” formula “)”
formula :	? Textual Definition of FOL ?
TYPE :	? Types from ROS or custom type ?
STRING :	? Character String?

Table 1: A simplified description of RCL's syntax in Extended Backus-Naur form (EBNF). A “|” shows alternatives, quotation marks shows literals, a “?” shows zero or one instances, a “*” shows zero or more instances, a “+” shows one or more instances. Textual descriptions of a rule are enclosed in “?”.

We define three rules to map the values from a ROS message to the matching variable in an RCL contract. The rule that is used depends on the type of the RCL variable that the ROS message matches. These rules are suitable for most mappings but may not cover every situation, so bespoke rules can be added to capture more complicated mappings. We discuss the application of these mapping rules for our case study in § 4.2.

The simplest rule is the **Scalar** rule, which is applied when the RCL variable is a scalar type. The rule maps a single variable in a ROS topic to a single RCL variable of the same type. For example, if the ROS topic and RCL variable are both integers, we can map one directly to the other.

The **Collection** rule applies when the RCL variable is a collection type (a set, sequence, or tuple) containing constants. If the ROS message is a single value, then it should map to one of the elements in the collection. We use this to map ROS strings into RCL, for example if the ROS message is a string such as “stop” or “go” **Collection** can map this into a constant in RCL if the contract defines the variable's type as $\{stop, go\}$. Because the contracts assume that the variable is drawn from the given collection, it is left for the developer to decide how to handle situations where the value of the ROS message is not present in the collection.

The **Function** rule applies when the RCL variable is a function. The rule maps the values of the variables in the ROS message to the function's parameters. If there is a mismatch between the number of variables in the ROS message and the number of parameters in the RCL function, then we map the variables from the ROS message that match the types of the function parameters in the order that they appear in the message. For example, if the ROS message contains two variables `int16, int16` and the RCL function is $\mathbb{N} \times \mathbb{N} \to \mathbb{B}$, then **Function** applies the **Scalar** rule to map the first `int16` variable to the first $\mathbb{N}$ parameter and the second `int16` variable to the first $\mathbb{N}$ parameter. However, if the ROS message has three variables `string, int16, int16` then **Function** will ignore the `string` variable and apply **Scalar** to map the two *int16* variables as before.

Each assumption and guarantee is declared separately, in a plain-text version of FOL, which uses keywords such as `forall` and `in` to represent logical operators. Table 1 shows a simplified Extended Backus-Naur form (EBNF) grammar for an RCL contract. The “TYPE” rule defines the built-in ROS message types and allows custom types to be defined. Listing 2 (§4.2) shows one of the RCL contracts that we used during this work.

VANDA[7] parses RCL files, synthesises executable monitors (§3.6), and can produce LaTeX versions of the contracts. It is written in Python3 and uses the Lark parsing library[8]. Fig. 2 shows the steps taken (and files involved) in the synthesis of a monitor from an RCL contract. This process comprises the following three steps. (1) **parse**: parse a contract file and, if the contract is well-formed, produce a parse tree. (2) **extract**: pre-process the parse tree from Step 1 to extract the node name, topics and, guarantees into a `Contract` object. (3) **translate**: a *contract translator* uses the `Contract` object from Step 2 to produce the configuration file and monitor structure. The guarantees are translated by a *FOL translator* class, which is a Lark `Interpreter` that converts the FOL parse tree into the monitoring language (§3.6).

Note that the current implementation of Vanda extracts the node name, topics, and guarantees for monitor synthesis. Assumptions remain part of the RCL contract and are central to the compositional reasoning in Step 3, where the calculus checks if a node's assumptions are entailed by the guarantees of the nodes that provide its inputs. Consequently, many assumptions do not need to be monitored separately because they are either trivially true, supplied by the environment, or already represented as guarantees of other nodes. Monitoring only guarantees also reduces runtime overhead, since the generated ROSMonitoring monitors need to evaluate fewer conditions at runtime.

VANDA is built so as to make it easy to change the input and output formats. The input language is defined in a Lark grammar, so that we can update RCL if needed. The translation is implemented by two files (the contract translator and FOL translator mentioned previously), so updating VANDA to produce monitors in a different formalism should be relatively straightforward.

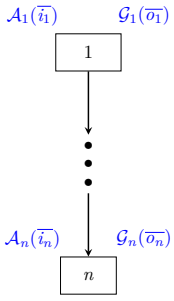
So far, we have described how contracts for individual nodes are expressed in FOL and discussed how these are encoded in RCL. Next we present our calculus for reasoning about how specifications for individual nodes are combined.

3.4 Calculus for Combining Node Specifications

Nodes in a (modular) system can be linked as long as their input types and requirements match. The basic way to describe these structures is to first have the contract capture all input and output variables and then describe how they are combined using suitable inference rules. We compose the contracts of individual nodes in a number of ways, the simplest being sequential composition:

R1:

$$\frac{\begin{array}{l}\forall\, \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_1(\overline{o_1}) \\ \vdots \\ \forall\, \overline{i_n}, \overline{o_n} \cdot \mathcal{A}_n(\overline{i_n}) \;\Rightarrow\; \Diamond \mathcal{G}_n(\overline{o_n}) \\ \overline{o_1} = \overline{i_2} \wedge \ldots \wedge \overline{o_{n-1}} = \overline{i_n} \\ \vdash \Big(\forall\, \overline{o_1}, \overline{i_2} \cdot \mathcal{G}_1(\overline{o_1}) \;\Rightarrow\; \mathcal{A}_2(\overline{i_2}) \\ \qquad\qquad \wedge \\ \qquad\qquad \vdots \\ \qquad\qquad \wedge \\ \qquad \forall\, \overline{o_{n-1}}, \overline{i_n} \cdot \mathcal{G}_{n-1}(\overline{o_{n-1}}) \;\Rightarrow\; \mathcal{A}_n(\overline{i_n}) \Big)\end{array}}{\Big(\forall\, \overline{i_1}, \overline{o_n} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_n(\overline{o_n}) \Big)}$$



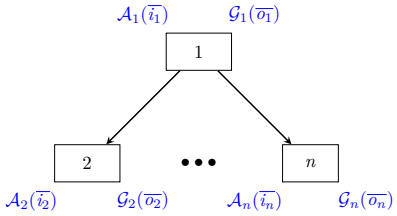
Rule **R1** applies to nodes that are connected in a *linear sequence*, where the output of one is the input of the next, etc. If the guarantee of the first implies the assumption of the second etc. then, if the assumption of the first node holds, we can conclude that the guarantees of the final node in the sequence will eventually hold. This rule provides a basic starting point, however, robotic systems are generally more complex than this, often including multiple, branching outputs.

[7] VANDA, which means 'Oath' in Quenya, is available at: `https://github.com/autonomy-and-verification/ros-contract-language/tree/v0.3.1-ras`. Accessed: 16/01/2026

[8] Lark parsing library: `https://github.com/lark-parser/lark` Accessed: 03/11/2023

R2:

$$\frac{\begin{array}{l}\forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_1(\overline{o_1}) \\ \qquad\qquad \vdots \\ \forall \overline{i_n}, \overline{o_n} \cdot \;\mathcal{A}_n(\overline{i_n}) \;\Rightarrow\; \Diamond \mathcal{G}_n(\overline{o_n}) \\ \overline{o_1} = \overline{i_2} \cup \ldots \cup \overline{i_n} \\ \vdash \Big(\forall \overline{o_1}, \overline{i_2}, \ldots, \overline{i_n} \cdot \;\mathcal{G}_1(\overline{o_1}) \Rightarrow \;\mathcal{A}_2(\overline{i_2}) \\ \qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad \wedge \qquad\quad \mathcal{A}_n(\overline{i_n}) \Big)\end{array}}{\begin{array}{l}\Big(\forall \overline{i_1}, \overline{o_2}, \ldots, \overline{o_n} \cdot \mathcal{A}_1(\overline{i_1}) \\ \quad \Rightarrow\; \Diamond \mathcal{G}_2(\overline{o_2}) \wedge \ldots \wedge \Diamond \mathcal{G}_n(\overline{o_n}) \Big)\end{array}}$$

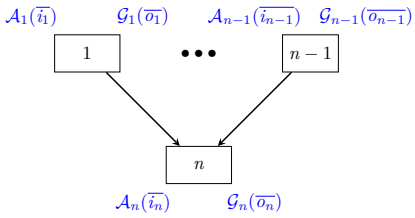


Rule **R2** is used when a node has *branching outputs*. Here, the combined inputs of all of the 'leaf' nodes is equal to the output of the 'root' node, while the guarantee of the 'root' node implies the assumptions of each 'leaf' node, as shown above (where we assume that the union operator, $\cup$, can takes vectors as parameters and merge them in the same way as with sets).

Rule **R3** deals with the converse architecture, where a node's input is the *union of several outputs* from other nodes. As expected, this rule is essentially the dual of **R2**. Note that there is a simplifying assumption here: that the outputs persist once generated, according to the $\Diamond$ constraint. Under this assumption, all the required inputs (from nodes 1 to $n-1$) will be available at the same time. In future work, we will consider weakening this assumption, thus adding further timing constraints. We assume that a user of this calculus ensures that this simplifying assumption is met in order to use the rules.

R3:

$$\frac{\begin{array}{l}\forall \overline{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\overline{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_1(\overline{o_1}) \\ \qquad\qquad \vdots \\ \forall \overline{i_n}, \overline{o_n} \cdot \mathcal{A}_n(\overline{i_n}) \;\Rightarrow\; \Diamond \mathcal{G}_n(\overline{o_n}) \\ \overline{i_n} = \overline{o_1} \cup \ldots \cup \overline{o_{n-1}} \\ \vdash \Big(\forall \overline{i_n}, \overline{o_1}, \ldots, \overline{o_{n-1}} \cdot \mathcal{G}_1(\overline{o_1}) \\ \qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad \wedge \\ \qquad\quad \mathcal{G}_{n-1}(\overline{o_{n-1}}) \;\Rightarrow \mathcal{A}_n(\overline{i_n}) \Big)\end{array}}{\Big(\forall \overline{i_1}, \ldots\; \overline{i_{n-1}}, \overline{o_n} \cdot \mathcal{A}_1(\overline{i_1}) \wedge \ldots \wedge \mathcal{A}_{n-1}(\overline{i_{n-1}}) \Rightarrow \Diamond \mathcal{G}_n(\overline{o_n}) \Big)}$$



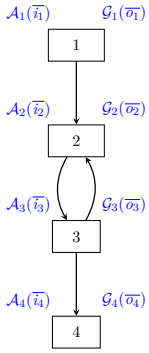
These three simple inference rules (**R1**, **R2** and **R3**) constitute our basic calculus for reasoning about node-level FOL contracts in a robotic system. We do *not* specify the fine-grained concurrency/streaming of processes/data but are just specifying the interface expectations of nodes in a robotic system.

We use the $\Diamond$ operator from FOTL to represent that some arbitrary time has passed. Using the $\Diamond$ operator gives us the flexibility to describe and refine a range of temporal aspects. We assume that contracts are neither mutually dependent, nor circularly dependant. However, we intend to investigate this in future work and potentially leverage existing techniques like [49].

We can extend **R1-3** with rule **R4**, below, which captures looping behaviour. We consider this rule to be an extension to our core calculus because it is directly derived from the first three rules.

R4:

$$\frac{\begin{array}{l}\forall \dot{i_1}, \overline{o_1} \cdot \mathcal{A}_1(\dot{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_1(\overline{o_1}) \\ \forall \dot{i_2}, \overline{o_2} \cdot \mathcal{A}_2(\dot{i_2}) \;\Rightarrow\; \Diamond \mathcal{G}_2(\overline{o_2}) \\ \forall \dot{i_3}, \overline{o_3} \cdot \;\mathcal{A}_3(\dot{i_3}) \;\Rightarrow\; \Diamond \mathcal{G}_3(\overline{o_3}) \\ \forall \dot{i_4}, \overline{o_4} \cdot \;\mathcal{A}_4(\dot{i_4}) \;\Rightarrow\; \Diamond \mathcal{G}_4(\overline{o_4}) \\ \overline{i_2} = \overline{o_1} \cup \overline{o_3} \wedge \overline{i_3} = \overline{o_2} \wedge \overline{o_3} \subseteq \overline{i_4} \\ \vdash \forall \overline{i_2}, \overline{o_3} \cdot \mathcal{A}_2(\dot{i_2}) \Rightarrow \Diamond \mathcal{G}_3(\overline{o_3}) \\ \vdash \forall \overline{i_3}, \overline{o_2}, \overline{o_4} \cdot \mathcal{A}_3(\dot{i_3}) \;\Rightarrow\; \Diamond \mathcal{G}_2(\overline{o_2}) \wedge \Diamond \mathcal{G}_4(\overline{o_4}) \\ \vdash \forall \overline{i_1}, \overline{i_3}, \overline{o_2} \cdot \mathcal{A}_1(\dot{i_1}) \wedge \mathcal{A}_3(\dot{i_3}) \;\Rightarrow\; \Diamond \mathcal{G}_2(\overline{o_2})\end{array}}{\left(\forall \overline{i_1}, \overline{o_4} \cdot \mathcal{A}_1(\dot{i_1}) \;\Rightarrow\; \Diamond \mathcal{G}_4(\overline{o_4})\right)}$$



Observe that the three lines with the $\vdash$ symbol are a direct result of applying **R1**, **R2** and **R3**, respectively, to the nodes in the loop illustration above. The conclusion below the line is drawn from their combination and simplification. Although not part of our core calculus, we list **R4** here to explicitly capture the behaviour of loops and it can easily be extended to incorporate an arbitrary number of nodes between nodes 2 and 3 in the loop illustration above. Alternatively, we could augment **R4** to include fixed-point reasoning in first-order temporal logic [50]. It is well-known that reasoning over loops can be complex, typically requiring the addition of loop invariants that are not always obvious or straightforward to define [51]. In the derivation above, we assume that all of the nodes are verified to meet their assumptions and guarantees and that they terminate, producing a correct result for each given input. When it comes to automating these derivation rules we may need to consider how to define appropriate, and ideally generic so that they can be easily instantiated, loop invariants but this is left as future work.

Our inference rules describe how node contracts should be interpreted when multiple nodes are combined in various ways. These rules are simple and therefore transparently sound. For brevity, we omit the soundness proofs here because they do not add to the novelty of this paper. It is a separate verification step to demonstrate that the individual node itself obeys its own contract and we discuss this in more detail in the next subsection. It is through this node verification process that users verify that whenever the assumptions for a particular node are true, then the guarantee should hold (after execution). In this way, our calculus, in practice, will avoid situations where trivially-true implication statements occur. Our calculus is thus to be used only with nodes that have been verified to preserve their guarantees when their assumptions are satisfied by the environment.

At present, our calculus is applied manually and we intend to investigate future ways to automate its application. This includes incorporating useful tactics for simplifying FOL formulas as needed. For this, we intend to incorporate a first-order theorem proving approach such as Vampire [52].

3.5 Guiding Heterogeneous Verification

Our RCL contracts provide a high-level specification of the system's requirements. These *node-level* contracts are used to *guide* the verification of individual nodes, because a formal link is often difficult or impossible. However, the FOL specification, and other information in the contract, can be used to inform the (formal or non-formal) verification process.

Each node can present its own challenges when verifying that it obeys its FOL contract, so the most suitable verification method should be chosen for each node. Some nodes may be amenable to formal verification, such as an agent that can be model checked for correctness properties (as in our previous work [19]). Other nodes may be based on neural networks, e.g. a vision classifier that might need a specific testing approach [53]. Some nodes might be more critical to the system's safety requirements than others, these nodes are likely to be the focus of the most robust formal verification (as we suggest in [54, 55]). Our approach leaves whoever is verifying the node to choose the most suitable verification method(s), since they are best placed to make this decision.

We use the contract's high-level specification as a guide, or a target for what properties a particular node should obey, supporting developing the system from abstract specifications. Specifying what a system should do is often the most time-consuming part of formal verification [56], so reusing a specification throughout the development process makes that initial effort more worthwhile.

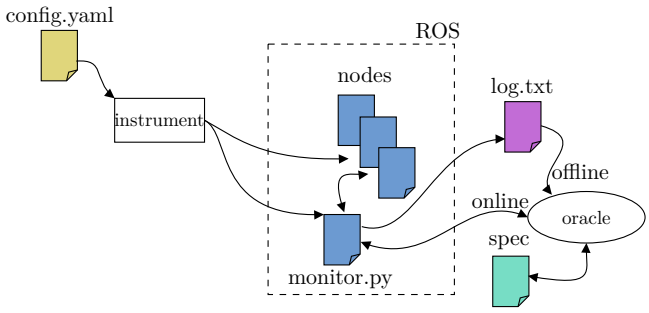


Figure 3: High-level overview of ROSMonitoring [57].

When developing a system from an abstract specification, a developer can start by specifying a high-level contract for a node's basic behaviour, and then verify more concrete properties about the node's functionality using their chosen verification method. For example, as in [28], a high-level contract for a planner might require that all of the plans it produces are obstacle-free, but the detailed planner verification might also verify that all points in the plan are valid (e.g. within the map), that the planner does not deadlock, or that the plan conforms to some measure of optimality.

Crucially, our approach enables a system to be verified using a range of verification techniques without needing to decide beforehand which techniques will be used. As mentioned in §1, using a variety of heterogeneous verification techniques on one system can be beneficial, particularly in the robotics domain [10, 14, 19, 28, 43].

The path from an RCL contract to *formal* verification is fairly clear, as we describe in §4. Some formal methods are also based on FOL and sets, so the guarantees in the contracts can be checked directly. Whereas, for other methods, more effort may be needed to capture the contract's properties in its specification language. In either case, the contract describes the properties that the chosen formal method must check.

The path from an RCL contract to *non-formal* verification is made easier because each contract provides an unambiguous specification of a node's requirements. A suitably skilled test engineer should be able to, for example, create software tests, simulation scenarios, etc.; that check the node for the properties described in the guarantee. Having a formal specification provides validation that the 'right' properties are being verified.

3.6 A Runtime Verification Safety-Net

The final part of our approach is to introduce a *safety net* of RV monitors, one for each RCL contract. Each monitor compares a trace of events produced by the system, with a formal model of the intended behaviour. This RV safety net ensures that the nodes conform to their contracts and, consequently, support the inferred system-level properties.

We use the formalism-agnostic general-purpose framework, ROSMonitoring [57], which is built for RV of ROS systems. The monitors used by ROSMonitoring follow a standard publish/subscribe pattern. Each monitor is a ROS node that subscribes to the topics needed to observe the behaviour that is relevant to its property, and publishes a message to inform the system that a violation was observed. We chose ROSMonitoring because its monitors can be easily distributed through the system to check the contracts for each node, and it can be used with various versions of ROS/ROS2.

This work uses ROSMonitoring 1.0 [57], which monitors ROS programs only through the topics they communicate over. As mentioned in §3.3, the impact of this limitation is that the RCL contracts can only state what topics their inputs and outputs correspond to, because this information is used during the synthesis of ROSMonitoring monitors. However, this means that RCL contracts cannot link their inputs and outputs to services, though services are often used in ROS programs. An updated version of ROSMonitoring (2.0) [58] introduces support for monitoring services, providing a route to enabling our approach to do the same. Updating RCL and Vanda to cater to ROS systems using services and actions is left as future work, and discussed in §6.

From RCL topics to ROSMonitoring configuration file

ROSMonitoring takes a configuration file as input ('config.yaml' in Fig. 3) where the user specifies the nodes and the topics that will be analysed by the monitors. Our tool, VANDA, automatically generates a configuration file from an RCL specification. Generating the configuration file only requires the name of the topics to monitor and their message types. An example of a configuration file is shown later when we describe the case study (§4.5).

Given a configuration file, ROSMonitoring generates one (or multiple) runtime monitor(s) and changes the ROS system nodes to allow the monitors to intercept the message exchanges needed for the verification. The remaining step is to produce the *Oracle*, the component that performs the formal verification.

From RCL guarantees to an RML Oracle

In an assume-guarantee contract, the guarantee of a component is conditional on its assumptions being satisfied. However, for connected software nodes, our calculus checks that the assumptions of a receiving node follow from the guarantees of the producing node(s). In such cases, monitoring the receiving node's assumption in addition to the producing node's guarantee would duplicate runtime checks and increase overhead. We therefore focus on guarantee monitoring, which provides a low-overhead safety net for detecting when software components fail to produce outputs that satisfy their contracts.

This choice has implications for debugging and failure diagnosis. A guarantee violation identifies where the monitored contract is violated, but the root cause may be an unmet assumption. When the unmet assumption is due to another software node producing incorrect information, our approach still captures this, since the producing node's guarantee is the source of the failure. However, assumptions that correspond directly to environmental conditions may require explicit monitoring to distinguish environmental failures from software failures. Extending Vanda with assumption monitors and richer verdicts that distinguish assumption violations from guarantee violations is therefore a useful direction for future work.

The `Oracle` (shown in Fig. 3) verifies that a system's trace obeys the formal specification, producing a *verdict* as output. In ROSMonitoring, the Oracle is decoupled from the monitoring framework and can be specified with any formalism, here we have chosen the Runtime Monitoring Language (RML) [59] because the translation from RCL to RML was the most intuitive and direct.

An RML property is a tuple $\langle t, ETs \rangle$, with a term t , and a set of event types $ETs = \{ET_1, \ldots, ET_n\}$. An event type ET is represented as a set of pairs $\{k_1 : v_1, \ldots, k_n : v_n\}$, where each pair identifies a specific piece of information (k_i) and its value (v_i). Given an event type ET, an event Ev, denoted as a set of pairs $\{k'_1 : v'_1, \ldots, k'_m : v'_m\}$ as well, matches ET if $ET \subseteq Ev$, which means $\forall (k_i : v_i) \in ET \cdot \exists (k_j : v_j) \in Ev \cdot k_i = k_j \wedge v_i = v_j$. In practice, an event type ET specifies the requirements that an event Ev must satisfy to be considered valid. For instance, an event type ET could be $\{pos : (waypoint1)\}$, meaning that all events containing pos with value $(waypoint1)$ are valid. An event Ev generated by a moving robot could be $\{speed : 1.6, pos : (waypoint1)\}$, meaning that the robot is moving at speed 1.6 [m/s], and is currently at position $(waypoint1)$.

An RML term t can be:

- ET, denoting a singleton set containing the events Ev s.t. $ET \subseteq Ev$
- $t_1\ t_2$, denoting the concatenation of two sets of traces
- $t_1 \wedge t_2$, denoting the intersection of two sets of traces
- $t_1 \vee t_2$, denoting the union of two sets of traces
- $\{let\ x;\ t'\}$, denoting the set of traces t' where the variable x can be used
- $t'*$, denoting a chain of concatenations of trace t'

where t_1, t_2 and t' are RML terms.

Event types can be negated. Given an event type ET, the term $\neg\ ET$ denotes its negation. Specifically, $\forall_{Ev} . ET \subseteq Ev \Leftrightarrow \neg\ ET \nsubseteq Ev$. In the rest of the paper, we also apply the notion of negation to the other RML terms. For instance, if the term is $ET_1 \wedge ET_2$, its negation is $\neg\ ET_1 \vee \neg\ ET_2$; and the same reasoning is applied for the other operators.

Event types can contain variables. Considering the previous event type, we could have its parametric version as $ET(x) = \{pos : (x)\}$, where we do not force any value for the waypoint. This event

$$\text{(atom)}\ \frac{}{atom \to \langle atom, \{\}\rangle} \qquad \text{(and/or)}\ \frac{g_1 \to \langle t_1, ET_1\rangle \wedge g_2 \to \langle t_2, ET_2\rangle}{g_1\, op\, g_2 \to \langle t_1\, op\, t_2, ET_1 \cup ET_2\rangle}\ op \in \{\wedge, \vee\}$$

$$\text{(iff)}\ \frac{g_1 \to \langle t_1, ET_1\rangle \wedge g_2 \to \langle t_2, ET_2\rangle}{g_1 \Leftrightarrow g_2 \to \langle (t_1 \wedge t_2) \vee (\neg\, t_1 \wedge \neg\, t_2), ET_1 \cup ET_2\rangle}$$

$$\text{(implies)}\ \frac{g_1 \to \langle t_1, ET_1\rangle \wedge g_2 \to \langle t_2, ET_2\rangle}{g_1 \Rightarrow g_2 \to \langle (\neg\, t_1 \wedge t_2) \vee (\neg\, t_1 \wedge \neg\, t_2) \vee (t_1 \wedge t_2), ET_1 \cup ET_2\rangle}$$

$$\text{(equals)}\ \frac{}{a_1 == a_2 \to \langle ET, \{\langle ET, (a_1 : a_2)\rangle\}\rangle}$$

$$\text{(not-equals)}\ \frac{}{a_1 != a_2 \to \langle \neg\, ET, \{\langle ET, (a_1 : a_2)\rangle\}\rangle}$$

$$\text{(exists)}\ \frac{g \to \langle t, ET\rangle}{\exists\, x \cdot g \to \langle \{let\ x;\ t\}, ET\rangle}\ \text{(guarantee)}\ \frac{g \to \langle t, ET\rangle}{G(g) \to \langle (t)*, ET\rangle}$$

$$\text{(guarantees)}\ \frac{g_1 \to \langle t_1, ET_1\rangle \wedge \ldots \wedge g_n \to \langle t_n, ET_n\rangle}{G(g_1), \ldots, G(g_n) \to \langle t_1 \wedge \ldots \wedge t_n, ET_1 \cup \ldots \cup ET_n\rangle}$$

Figure 4: RML partial translation.

type matches all events containing *pos* with any x value. When an event matches an event type with variables, the variables get the values from the event.

An example of using RML to define contracts, consider the term $ET_1 ET_2 \vee ET_3 ET_4$, with ET_i being some event type describing a certain kind of event (as before). However, differently from before, here, we have a union of two concatenations; which means the language recognised by this term is

$$\{Ev\ Ev' \mid ET_1 \subseteq Ev, ET_2 \subseteq Ev'\} \cup \{Ev\ Ev' \mid ET_3 \subseteq Ev, ET_4 \subseteq Ev'\}$$

Naturally, the same reasoning can be applied to the $\wedge$ operator; the only difference would be in using $\cap$ instead of $\cup$.

As previously mentioned, Vanda synthesises RML specifications from the RCL contracts. Here we describe how Vanda translates the RML guarantees into RML. In principle, both the assumptions *and* guarantees can be translated into RML, since both are expressed as RCL formulae. In the current implementation, Vanda translates only guarantees. This keeps the runtime monitoring layer lightweight and avoids duplicating checks that are already represented compositionally, since the assumptions of many nodes are entailed by the guarantees of the nodes that provide their inputs. Translating assumptions would follow a similar process but would be most useful for assumptions that correspond to external environmental conditions or for applications that require finer-grained runtime diagnosis.

Fig. 4 shows the operational semantics of the translation function from RCL descriptions to RML specifications that it implements. Here, each rule formalises a single-step translation. For instance, the (and/or) rule denotes how to translate the conjunction/disjunction of two RCL guarantees. This can be done first by translating the two guarantees g_1 and g_2 into RML; then, by combining the results obtained using the corresponding RML operator.

For the (equals) rule, at the guarantee level we require that a certain variable in a topic has a specific value. This is mapped into an event type that requires the event to have the same value. Recall that an event matches an event type if the latter is included in the former. Since we want to guarantee that the information identified by a_1 has value a_2, the resulting event type has to check that this holds, which can be done by adding the pair $a_1 : a_2$ in the event type.

The (exists) rule tackles the use of variables. In RCL, the contract says that there exists an x for which the guarantee g holds. This can be straightforwardly mapped into a parametric term in RML, where the variable x is used inside the term t (the one derived by g). As usual, the (forall) rule can be obtained by negating the (exists) rule. Finally, the (guarantee) and (guarantees) rules denote the translation step for single and multiple guarantees, respectively. To translate multiple guarantees, each guarantee g_i is translated separately, and the resulting RML terms t_i are combined using the $\wedge$ RML operator. In this way, all guarantees are required to be satisfied. To translate a guarantee $G(g)$, first g is translated into the RML term t. A $*$ post-fix operator is then added to t, meaning that t can be repeated as many times as necessary. We need this operator because the guarantees need to be checked continuously, not just once.

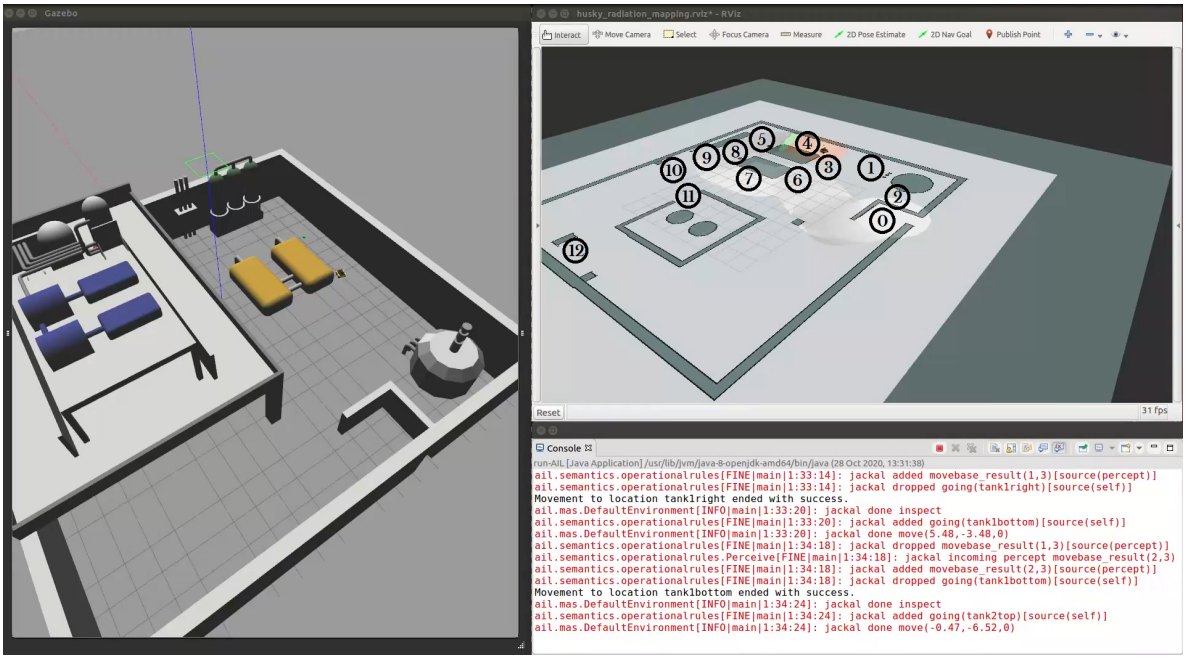


Figure 5: Simulation environment of the remote inspection case study. On the left-hand side is the Gazebo 3D simulation window. On the top right-hand side is the RViz window, a visualisation interface containing the map that the rover uses for path planning, a visual representation of radiation levels (green, orange, red) around the robot, and the waypoints from 0 to 12 that the robot must inspect. On the bottom right-hand side is the autonomous agent console with the log of its execution.

From RCL to ROSMonitoring monitors

The generation of monitors in ROSMonitoring from RCL guarantees requires one more additional step. The RCL specifications are based on an abstraction of the system, while the monitors in ROSMonitoring are deployed in the real system under execution. Therefore, to ensure the monitors will be able to capture all of the necessary events (execution traces) produced by the system, it is necessary to instrument the system with any missing events. For example, these missing events can be a mismatch of names used in the contracts versus what is used in the implemented system, different representations of data types, implicit information that is not directly being published in a ROS topic, etc.

4 Case Study: Remote Inspection

Our case study is a *Jackal*[9] rover performing remote inspection in a nuclear waste store, in a 3D Gazebo[10] simulation. The Jackal uses a (simulated) sensor to take radiation measurements at given waypoints. We adapted this case study from the simulation described in [60], adding an autonomous agent that makes the high-level decisions that control the Jackal.

The rover's goal is to inspect 12 waypoints that are located inside the simulated nuclear waste store (see Fig. 5). The autonomous agent decides which waypoint to inspect next, depending on the current radiation readings. It should always be aware of high radiation values, since this could put the robot in danger, as well as indicating a problem in the waste store that requires further investigation. The rover's battery is not simulated, so we assume that the rover has enough power for the duration of the scenario. We use the ROS *move base* package for low-level path planning.

At a high level the system should obey the following requirements.

REQ1: The robot should inspect each waypoint as long as the radiation level at a waypoint is not high ("red").

[9] Jackal: `https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/` Accessed: 03/11/2023

[10] Gazebo Simulator: `http://gazebosim.org/` Accessed: 03/11/2023

REQ2: Each waypoint only needs to be inspected at least once.

REQ3: If the radiation level at a waypoint is too high, then the robot will abandon the mission and return to the entry point.

REQ4: Eventually, all waypoints will have been inspected; or the mission will have terminated in failure, due to high levels of radiation.

In the rest of this section, we describe how to apply our approach to the remote inspection case study, from the manual specification of the contracts to the automatic generation of runtime monitors. The simulation environment, contracts, and verification assets for this chapter are available from the Zenodo repository that accompanies this paper: `https://doi.org/10.5281/zenodo.6941344`.

4.1 Step 1: Creating a System Model

In this section we describe how we generated the system model for the remote inspection case study. Fig. 6 shows the ROS graph of the remote inspection robotic system (minus the autonomous agent, because it is not programmed using ROS). As described in §3, we often find it helpful to adapt the ROS graph to produce a system model that is more amenable to analysis and verification.

Here, most of the nodes are well-tested off-the-shelf libraries, such as the Jackal *velocity controller* and the *move base* library. The former relates to the Jackal's internal velocity controllers, while the latter provides a suite of path planners. Both relate to the robot's navigation, and thus we abstract them into a *compound* node called **Navigation**. Similarly, we merge the nodes *imu* and *odometry*,into the **Localisation** *compound* node. We obtain the **Radiation Sensor** node in a similar way. Finally, because the autonomous agent is external to the ROS system, we create the **Agent** node as an abstraction of its implementation. Thus, the system model is composed of these four abstracted nodes, shown in Fig. 7 and described below.

Localisation: abstracts the Adaptive Monte Carlo Localisation (AMCL) ROS package[11], which uses sensor input to perform localisation. This is a well established package, so we do not define a contract for its specifics. Instead, it is enough (for our purposes) to know that the node will output what it believes to be the robot's position. The **Localisation** node takes input from hardware (a variety of sensors), but our contracts focus on the links between the *software* nodes; so it is enough for our contracts to specify its input as $\emptyset$ (defined as *SensorsType* in the specification's `context` clause, because the inputs and outputs are name-type pairs), knowing that the node will be able to access the sensor information.

Navigation: abstracts the well known *move base* package, which performs low-level path planning. Its inputs are the estimated position from the **Localisation** node, and the command from the **Agent** node to move to a particular position. It outputs a function mapping coordinates to a boolean, which is `true` if the Robot is at that position.

Agent: the most critical node to verify; because it makes the high-level decisions for the robot, and its implementation is not based on preexisting ROS packages. The agent is implemented in the agent programming language Gwendolen [61]. Its inputs are the output of the **Navigation** and **Radiation Sensor** nodes. It outputs a high-level command (either move or inspect).

Radiation Sensor: takes as input a radiation measurement r from its internal sensor and the command from the **Agent** to inspect a particular waypoint i. It outputs a confirmation that a particular waypoint has been inspected.

Once the model has been distilled, the next step is to define contracts for the individual nodes.

4.2 Step 2: Specifying Node Contracts

This step builds the contracts used to guide the heterogeneous verification and to enable synthesis of the runtime monitors. Each contract is written in RCL (described in §3.3), and consists of a node's inputs, outputs, assumptions, and guarantees (which are all written in a plain-text encoding of FOL). A contract also links the inputs and outputs to the topics in the underlying ROS program.

[11] Adaptive Monte Carlo Localisation package: `http://wiki.ros.org/amcl` Accessed: 03/11/2023

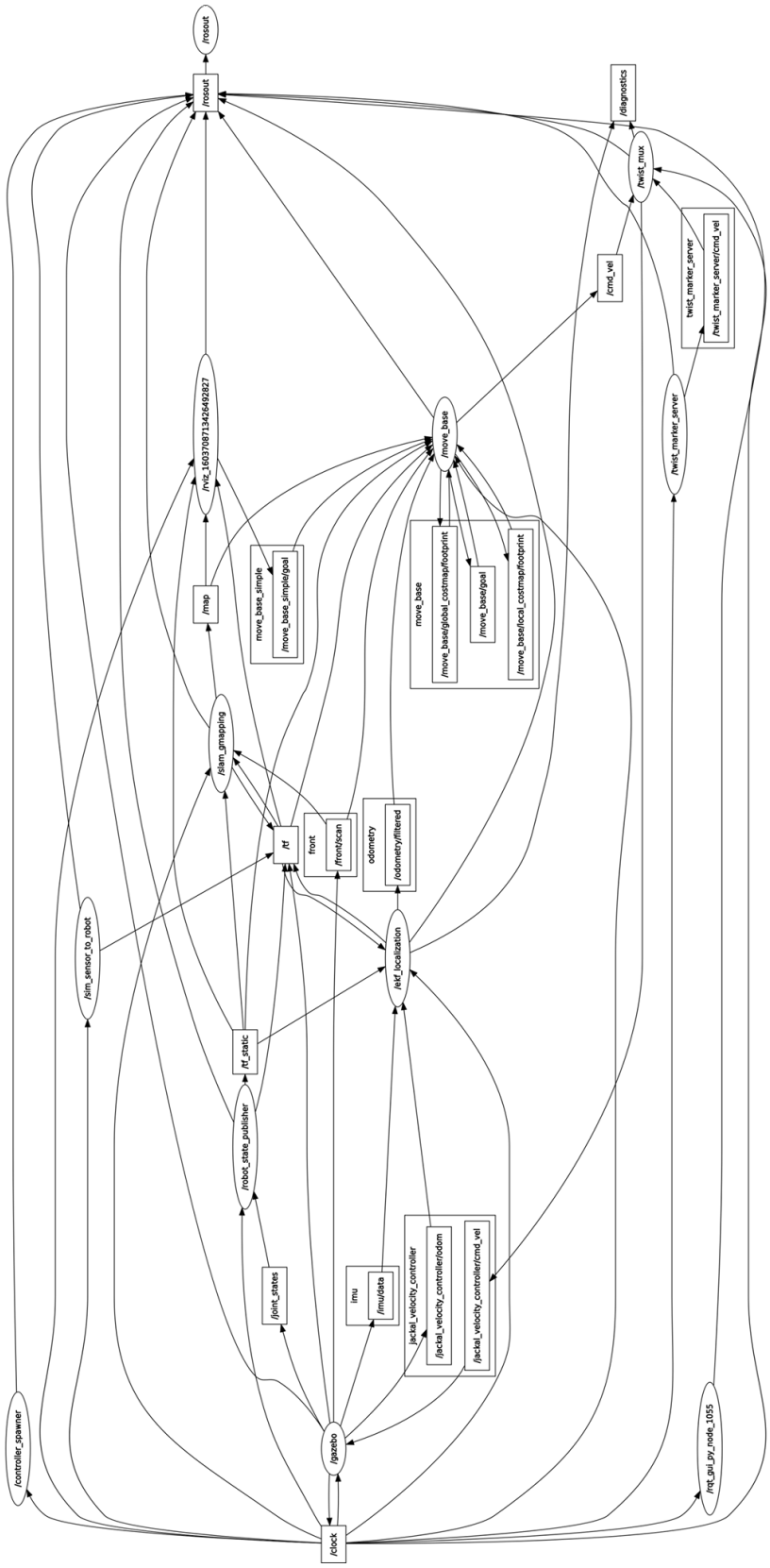


Figure 6: ROS graph automatically generated for the remote inspection case study. The ovals represent ROS nodes and the rectangles represent topics. The arrows connecting nodes and topics show the direction of messages.

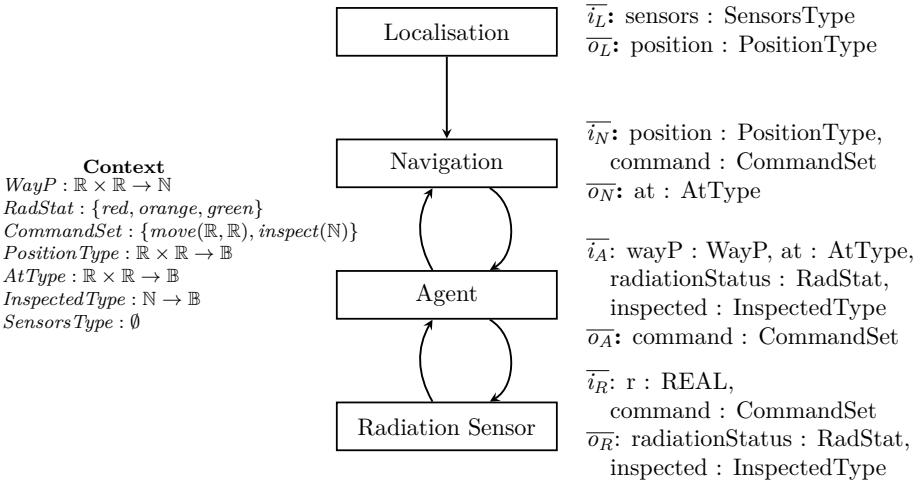


Figure 7: System model for the remote inspection case study. Arrows indicate data flow between the nodes.

Localisation

Input	**Text**	Input from hardware (a variety of sensors)
	FOL	$sensors : SensorsType$
Output	**Text**	The robot's estimated position
	FOL	$position : PositionType$
Assume	**Text**	N/A
	FOL	$TRUE$
Guarantees	**Text**	The node outputs a unique (x, y) coordinate that is the robot's estimated position
	FOL	$\exists! x, y \in \mathbb{R} \bullet out.position(x, y)$

Table 2: A summary of the **Localisation** node's inputs, outputs, assumptions, and guarantees.

As an example, Listing 1 shows the `context` clause that defines the types and constants used in the specification of our case study and Listing 2 shows the RCL contract for the **Agent** node. In the **Agent**'s contract, Listing 2, lines 3 and 5 identify the node's `inputs` and `outputs`, respectively.

Lines 7–11 show the ROS `topics` that the **Agent** uses and which input or output they correspond to (in the `matches` statement). For example, line 9 shows that the `at` field in the `gazebo_radiation_plugin/Snapshot` topic matches `in.at`. On line 14, the `assume` statement contains the assumption of the **Agent**'s contract. From line 16 onward, the `guarantee` statements contain a plain-text encoding of the **Agent**'s FOL guarantees. The links between the four nodes, and the inputs and outputs that link them, are shown in Fig. 7.

In §3.3 we describe the general rules that we uses to map ROS topics into RCL contract variables. Conceptually, the values from each ROS topic populates the RCL variable that it matches. However, it is important to note that the topics in the contracts are only used to synthesise the monitors, and there is no automatic connection between the contracts and the ROS program. We assume that there is a mapping that implements the rules, to link the more abstract specification in the RCL contracts with the concrete program.

Using Listing 2 as an example, we show how the rules map between the ROS program and the RCL contracts. The `in.inspected` and `in.at` variables are functions (as shown in Listing 1) so we apply the **Function** rule, which maps the values in the ROS message to the parameters of the function. For `in.inspected`, this maps an `int16` variable in the ROS message `gazebo_radiation_plugin/Snapshot inspected` to the `NATURAL` parameter of the function in RCL using the **Scalar** rule. For `in.at`, the ROS message contains two `float64` variables, so the **Scalar** rule is applied to each in turn, mapping them to the two `REAL` parameters of the `in.at`

function.

The `in.wayP` variable is also a function. Here the matching ROS topic (`gazebo_radiation_plugin/Snapshot wayPNow`) has three variables, but the `in.wayP` function only takes two parameters. For this mapping, the **Function** rule takes only the variables that match the types of the two `REAL` parameters of the function.

The `in.radiationStatus` variable's type is the set `{red, orange, green}` and the ROS topic contains only a single `string` variable. For this mapping we use the **Collection** rule, which maps the `string` variable to one of the three elements of the set. Given that we know the program only communicates strings "red", "orange", or "green", the verification steps assume this to be true. As a safety net, the runtime monitors (§4.5) would treat strings outside this set in the same way as "red" or "orange".

```
context{
 WayP : REAL x REAL --> NATURAL ;
 RadStat : {red, orange, green} ;
 CommandSet : { move(REAL, REAL), inspect(NATURAL) };
 PositionType : REAL x REAL --> BOOL ;
 AtType: REAL x REAL --> BOOL;
 InspectedType :  NATURAL --> BOOL;
 SensorsType : {}; }
```

Listing 1: The **Context** clause for the RCL contracts. This defines global types and constants.

```
node agent{

inputs( wayP : WayP, at : AtType, radiationStatus : RadStat, inspected : InspectedType )

outputs( command : CommandSet  )

topics( gazebo_radiation_plugin/Snapshot command matches(out.command),
gazebo_radiation_plugin/Snapshot inspected matches(in.inspected),
gazebo_radiation_plugin/Snapshot at matches(in.at),
gazebo_radiation_plugin/Snapshot wayPNow matches(in.wayP),
gazebo_radiation_plugin/Snapshot radiationStatus matches(in.radiationStatus) )

assume( in.radiationStatus in {red, orange, green} )

guarantee(forall(x', y' in REAL, i in NATURAL |
  in.at(x', y') == TRUE and in.wayP(x', y') == i and in.inspected(i) == TRUE and in.radiationStatus !
     in {red, orange} ->
  exists(x, y in REAL | (in.wayP(x, y) == i + 1  or
    (forall( x'', y'' in REAL | in.wayP(x'', y'') != i+1 and in.wayP(x, y) == 0))) and out.command ==
     move(x, y) )) )

guarantee(forall(x', y' in REAL, i in NATURAL |
  in.at(x', y') == TRUE and in.wayP(x', y') == i and in.inspected(i) == FALSE -> out.command ==
     inspect(i) ) )

guarantee(forall(x', y' in REAL, i in NATURAL |
  in.radiationStatus in {red, orange} or not exists( x, y in REAL | in.wayP(x, y) == i + 1 )
  -> exists (x'', y'' in REAL | in.wayP(x'', y'') == 0 and out.command == move(x'', y'') ) ) )
}
```

Listing 2: The **Agent**'s RCL contract.

Finally, to handle the mapping between the `out.command` variable and the `gazebo_radiation_plugin/Snapshot command` topic we add a bespoke rule, **CommandMapping**. The `command` topic contains a `string`, which is either "move" or "inspect"; and an `int16`, which is the id of a waypoint. The **CommandMapping** rule checks the value of the `string`. If the `string == "move"` then it uses the inverse of `WayP` (`WayP^-1`) to obtain the coordinates of the of the waypoint that the rover should move to, then uses the **Function** rule to map these two coordinate values to the parameters of the `move()` function. If the `string == "inspect"` then **CommandMapping** uses the **Scalar** rule to map the `int16` waypoint id to the `NATURAL` parameter of the `inspect()` function.

Tables 2, 3, 4, and 5 summarise the inputs, outputs, assumptions, and guarantees of each contract. The tables describe the statements in the contract in English, and then shows the relevant part of the contract. The FOL has been rendered with mathematical symbols for the convenience of the reader. The full contracts, in RCL, are shown in A.

The information contained in the contracts is useful for guiding both formal and non-formal verification of the nodes, as we describe later in §4.4. VANDA, the RCL tool, can also automatically synthesise runtime monitors from the contracts, which we describe in §4.5.

Navigation		
Input	**Text**	Robot's estimated position from the **Localisation** node, and the command from the **Agent** node
	FOL	$position : PositionType, \quad command : CommandSet$
Output	**Text**	Robot's position
	FOL	$at : AtType$
Assume	**Text**	There is a unique (x, y) coordinate position that represents the Robot's estimated position
	FOL	$\exists! x, y \in \mathbb{R} \bullet in.position(x, y) = TRUE$
Guarantees	**Text**	If the Robot is told to move to (x, y) and the **Localisation** node says that the Robot is at (x, y); then **Navigation** will conclude that it is at (x, y)
	FOL	$\forall x, y \in \mathbb{R} \bullet (\ (in.command = move(x, y)) \wedge (in.position(x, y) = TRUE)) \Leftrightarrow out.at(x, y) = TRUE$

Table 3: A summary of the **Navigation** node's inputs, outputs, assumptions, and guarantees.

In the next step, we use our calculus to derive system-level properties from the individual node contracts.

4.3 Step 3: Deriving System-Level Properties

In this section, we give an example of how our calculus (§3.4) works by presenting the manual derivation of one of the system-level properties from the contracts.

We start with the **Agent** node, which takes input from the **Navigation** and **Radiation Sensor**. Because **Agent** takes several inputs, we instantiate **R3**, the input union rule, to get D1:

$$\frac{\begin{array}{l}\forall \overline{i_N}, \overline{o_N} \cdot \mathcal{A}_N(\overline{i_N}) \ \Rightarrow \ \Diamond \mathcal{G}_N(\overline{o_N}) \\ \forall \overline{i_R}, \overline{o_R} \cdot \ \mathcal{A}_R(\overline{i_R}) \ \Rightarrow \ \Diamond \mathcal{G}_R(\overline{o_R}) \\ \forall \overline{i_A}, \overline{o_A} \cdot \ \mathcal{A}_A(\overline{i_A}) \ \Rightarrow \ \Diamond \mathcal{G}_A(\overline{o_A}) \\ \overline{i_A} = \overline{o_N} \cup \overline{o_R} \\ \vdash \ \left(\forall \overline{i_A}, \overline{o_N}, \overline{o_R} \cdot \mathcal{G}_N(\overline{o_N}) \wedge \mathcal{G}_R(\overline{o_R}) \Rightarrow \mathcal{A}_A(\overline{i_A}) \right)\end{array}}{\left(\forall \overline{i_N}, \overline{i_R}, \overline{o_A} \cdot \mathcal{A}_N(\overline{i_N}) \wedge \mathcal{A}_R(\overline{i_R}) \Rightarrow \Diamond \mathcal{G}_A(\overline{o_A}) \right)} \quad \text{(D1)}$$

To apply rule **R3**, we must show that we can deduce that the guarantees of the **Navigation** and **Radiation Sensor** nodes imply the assumption of the **Agent** node: $\forall \overline{i_A}, \overline{o_N}, \overline{o_R} \cdot \mathcal{G}_N(\overline{o_N}) \wedge \mathcal{G}_R(\overline{o_R}) \Rightarrow \mathcal{A}_A(\overline{i_A})$. For these three nodes, this is instantiated as:

$$\begin{array}{l}\forall \overline{i_A}, \overline{o_N}, \overline{o_R} \cdot [\forall x, y \in \mathbb{R} \cdot \\ \quad in.command = move(x, y) \wedge in.position(x, y) = true \Leftrightarrow out.at(x, y) = true] \\ \qquad \wedge [\forall i \in \mathbb{N} \cdot in.command = inspect(i) \Rightarrow (out.inspected(i) = true \wedge \\ \quad \wedge (0 \leq in.r < 120 \Rightarrow out.radiationStatus = green) \\ \quad \wedge (120 \leq in.r < 250 \Rightarrow out.radiationStatus = orange) \\ \quad \wedge (250 \leq in.r \Rightarrow out.radiationStatus = red))] \\ \quad \Rightarrow out.radiationStatus \in \{red, orange, green\}\end{array}$$

which is true by virtue of our definition of *radiationStatus*.

Applying **R3** here here allows us to conclude that if the nodes are correctly linked (by showing that the property after the $\vdash$ in D1 holds) then we can say that the correct input to the **Navigation** and **Radiation Sensor** nodes will result in the guarantee of the **Agent** being preserved. Thus, the system-level property that we derive is:

$$\forall \overline{i_N}, \overline{i_R}, \overline{o_A} \cdot (\exists! x, y \in \mathbb{R} \cdot position(x, y) = true) \wedge (0 \leq r) \Rightarrow \Diamond \mathcal{G}_A(\overline{o_A})$$

which tells us that when the robot is in a valid, unique position; and the observed radiation level is valid; then eventually the guarantee of the **Agent** will hold. This demonstrates that the **Agent**'s guarantee is dependent on the correct input and functioning of the **Navigation** and **Radiation Sensor** nodes.

The **Agent**'s guarantees (see Table 4) support the four requirements described in the introduction to this section. The **Agent** guarantees that that each waypoint will be inspected, unless dangerous

Agent		
Input	**Text**	Map of x, y pairs to waypoint ids, function that is `true` if the robot is at that position, and Radiation Status and inspected values from **Radiation Sensor**
	FOL	$wayP : WayP$, $at : AtType$, $radiationStatus : RadStat$, $inspected : InspectedType$
Output	**Text**	The command (move or inspect)
	FOL	$command : CommandSet$
Assume	**Text**	The Radiation Status that it receives is either red, orange, or green
	FOL	$in.radiationStatus \in \{red, orange, green\}$
Guarantees	**Text**	If the Robot is at, and has inspected waypoint i, and the radiation level was not dangerous, then it will move to the next waypoint or to waypoint 0 (the exit) if there is no next waypoint.
	FOL	$\forall x', y' \in \mathbb{R}, i \in \mathbb{N} \bullet ((in.at(x', y') = TRUE)$ $\land (in.wayP(x', y') = i) \land (in.inspected(i) = TRUE)$ $\land (in.radiationStatus \notin \{red, orange\}))$ $\Rightarrow (\exists x, y \in \mathbb{R} \bullet ((in.wayP(x, y) = i + 1$ $\lor (\forall x'', y'' \in \mathbb{R} \bullet in.wayP(x'', y'') \neq i + 1$ $\land in.wayP(x, y) = 0))$ $\land out.command = move(x, y)))$
	Text	And, if the Robot is at, and has not inspected, waypoint i, then inspect it.
	FOL	$\forall x', y' \in \mathbb{R}, i \in \mathbb{N} \bullet ((in.at(x', y') = TRUE)$ $\land (in.wayP(x', y') = i)$ $\land (in.inspected(i) = FALSE))$ $\Rightarrow out.command = inspect(i)$
	Text	And, if the Robot inspects a waypoint and the radiation level is dangerous, or there are no more waypoints, then return to the original waypoint (the exit)
	FOL	$\forall x', y' \in \mathbb{R}, i \in \mathbb{N} \bullet$ $((in.radiationStatus \in \{red, orange\})$ $\lor (\neg \exists x, y \in \mathbb{R} \bullet in.wayP(x, y) = i + 1))$ $\Rightarrow (\exists x'', y'' \in \mathbb{R} \bullet (in.wayP(x'', y'') = 0)$ $\land (out.command = move(x'', y'')))$

Table 4: A summary of the **Agent** node's inputs, outputs, assumptions, and guarantees.

levels of radiation are detected by the **Radiation Sensor** (REQ1 and REQ4); that a waypoint is only inspected if it hasn't been inspected yet (REQ2); that if the radiation level at a waypoint is too high, then it moves to the original waypoint (REQ3 and REQ4); and that when there are no more waypoints to inspect, the robot will return to the original waypoint (REQ1 and REQ4).

4.4 Step 4a: Heterogeneous Verification

This section describes our verification of the four nodes in our remote inspection case study. Each node presents its own verification challenges, and we have chosen a suitable approach to verify that each node obeys its contract. As mentioned in §3.5, the link between the contracts and the verification steps is *not* formal. The information in the contracts informs the verification steps, enables the use of either formal or non-formal methods as best suits the node being verified.

4.4.1 Localisation Node

The **Localisation** node's contract specifies that it should output a unique position ($\exists! x, y \in \mathbb{R} \cdot Position(x, y)$). Using code review, we checked that noise from the sensors did not change the position estimate if the robot has not moved. In practical terms, it might be necessary to allow a short time-window in which the node could obtain several sensor readings before converging on a single position estimate.

Radiation Sensor

Input	**Text**	The radiation value from the sensors, and the command from the **Agent**
	FOL	$r : \mathbb{R}, command : CommandSet$
Output	**Text**	Radiation Status and if this waypoint has been inspected or not
	FOL	$radiationStatus : RadStat, \quad inspected : InspectedType$
Assume	**Text**	The radiation reading is less than or equal to 0
	FOL	$0 \leq in.r$
Guarantees	**Text**	If the command received was to inspect waypoint i, then eventually waypoint i will be inspected; and the radiation at waypoint i will be categorised as either green, orange, or red
	FOL	$\forall i \in \mathbb{N} \bullet (in.command = inspect(i))$ $\Rightarrow$ ($(out.inspected(i) = TRUE)$ $\wedge$ $(0 \leq in.r \wedge in.r < 120 \Rightarrow$ $out.radiationStatus = green)$ $\wedge$ $(120 \leq in.r \wedge in.r < 250 \Rightarrow$ $out.radiationStatus = orange)$ $\wedge$ $(250 \leq in.r \Rightarrow$ $out.radiationStatus = red)$)

Table 5: A summary of the **Radiation Sensor** node's inputs, outputs, assumptions, and guarantees.

It would have been temping to verify this property experimentally, as we do for the **Navigation** node (see §4.4.2). However, the 3D Gazebo simulation of the nuclear waste store does not simulate noisy sensors (*i.e.*, in any given position the value returned by the simulated sensors was deterministic). Therefore, the position estimates do not change and while we could have verified this, it would not necessarily have told us anything useful.

Code review was chosen because of the previously mentioned limitations of the simulation and because the **Localisation** node relies on well-used ROS libraries. We inspected the code in the **Localisation** node and observed that the message specification for the node could only return one position at a time.

While a code review was enough for the purposes of this paper, stronger demonstrations may be needed for (*e.g.*) regulatory approval. For example, the performance of the **Localisation** node could be tested on a physical robot operating in a safe, test environment. We could specify a time window, after which a position estimate should have converged to a single value (or a set of values all within acceptable bounds of each other) and then test, using a number of routes around the mapped space, that this behaviour occurred.

4.4.2 Navigation Node

The **Navigation** node's contract specifies that if there was a command to move to a position (x, y) and the estimated current position of the rover is (x, y), then the rover has successfully arrived at its destination. We verified this experimentally in simulations.

Our verification used the 3D Gazebo simulation of the nuclear waste store as our test environment, and we created an autonomous agent that would take a random location within the waste store, navigate to that location and stop. We were then able to compare the goal location with the robot's actual position within the store, as reported by Gazebo.

The random location was generated as an action available to the agent in the Java environment that linked it to ROS. This used Java's built-in random number generation to select an x and a y coordinate within the the waste store map — random coordinates were regenerated if the original pair were in some inaccessible location, such as one of the tanks or pillars.

In total, we ran 47 experiments. On average the final location of the agent was 21.4cm away from the goal location with a standard deviation of 4mm. The worst result was a final position 43cm away from the goal location and the best result was 8cm from the goal location. The small standard deviation here implies that the simulated Jackal nearly always ends up approximately 21cm away from

its goal location. From a verification perspective, for this component to meet its contract we must decide whether a 21cm error counts as having "arrived at its destination". If this is within tolerance then the component has met its contract, if it is not then the component has not. In our case we had specifically configured `movebase` to have a 25cm tolerance for the controller in the x and y coordinates when achieving a goal, and so we can confirm that the node meets its contract.

Other Verification Approaches to Localisation, Mapping and Navigation

Robotic systems often contain a module for Simultaneous Localisation and Mapping (SLAM), which can be targeted by specific verification techniques [62, 63]. Given an estimate x (*i.e.*, a solution returned by a state-of-the art iterative solver), the verification approach evaluates whether x corresponds to a global optimum of a cost function $f(x)$. If the answer is positive, then the estimate can be trusted; if the answer is negative, then some recovery technique needs to be performed, because the estimate is not accurate and it is not safe to use. Moreover, this verification technique can be integrated seamlessly in standard SLAM pipelines, and provides a sanity check for the solution returned by standard iterative solvers. In [64] an improved extension builds upon the work from [62, 63] and introduces a novel formulation leading to a higher efficiency, reducing verification times by up to 50 times. SLAM relies on nonlinear iterative optimisation methods that in practice perform both accurately and efficiently. However, due to the non-convexity of the problem, the obtained solutions come with no guarantee of global optimality and may get stuck in local minima.

Other approaches targeting SLAM systems include [65, 66], which are more experimental works that focus more on testing rather than verification of SLAM. Also, in [67], a mathematical model is developed for SLAM verification and for physical model operation in the environment.

Our Localisation and Navigation nodes combine a SLAM capability with a navigation capability, without an explicit SLAM module so these approaches were not appropriate here.

4.4.3 Agent Node

The **Agent** node makes the high-level decisions for the robot. It is implemented in the Gwendolen [61] agent programming language and we use the agent-program model checker Agent Java PathFinder (AJPF) [68, 69] to verify its decisions. AJPF is an extension of Java PathFinder (JPF) [70] that enables formal verification of Belief-Desire-Intention (BDI) [71] agent programs by providing a property specification language that extends LTL with BDI constructs.

In the BDI model, agents use their *beliefs* (information that the agent believes about the world) and *desires* (a goal state that the agent wants to achieve) to select an *intention* for execution. To verify that the **Agent** node obeys its contract, we encode its guarantees into the property specification language for AJPF and check that the agent program meets these specifications.

The main parts of the Gwendolen code for the **Agent**[12] are shown in Listing 3. The agent starts with a list of static beliefs, such as *location_coordinate* that takes as parameters a location's numerical identifier (`Location` variable), name, and its map coordinates; and *next_location*, with two location identifiers as parameters, where the second parameter is the next location to visit after the first parameter.

The `inspect(Location)` plan (line 1) is triggered by the addition of the goal `inspect`. When the system starts, the **Agent** begins with a goal to inspect location `1`. The guard (*e.g.*, context or pre-conditions) of the plan is expressed inside the curly brackets. Here, the guard is that the **Agent** does not have the beliefs `danger_red`, `danger_orange`, or `going(0)`; where the first two beliefs indicate that there is radiation in the current location, and the latter is a *bookkeeping* belief used to track the location that the robot is moving towards (*0* indicates the entrance of the nuclear waste store, which is also the decontamination zone). The last belief, `location_coordinate(Location, LocationName, X, Y, Z)`, is a query to the belief base which will use the `Location` value (obtained from calling the plan, *e.g.*, `1` when the system starts) to match with its respective belief in the belief base and in turn unify the remaining open variables (*e.g.*, search for a `location_coordinate` belief where the `Location` term is 1, and then unify the remaining open variables with the values from the matched belief). If the guard test is successful, then the plan body (preceded by $<-$) is

[12]The specification is available in the Zenodo repository: `https://doi.org/10.5281/zenodo.6941344` Accessed: 16/01/2026

```
1  +!inspect(Location) : { ~B danger_red, ~B danger_orange, ~B going(0),
2                          B location_coordinate(Location,LocationName,X,Y,Z) }
3   <- +going(Location), move(X,Y,Z);
4
5  +movebase_result(Id,3) : { B going(L1), B next_location(L1, L2) }
6   <- -going(L1), inspect, +inspected(L1), +!inspect(L2);
7  +movebase_result(Id,3) : { B going(0) }
8      <- print("Decontamination"), do_nothing;
9  +movebase_result(Id,2) : { B going(Location) }
10     <- print("Failure");
11
12 +danger_red : { ~B going(0), B location_coordinate(0,door,X,Y,Z) }
13     <- +going(0), move(X,Y,Z);
14 +danger_orange : { ~B going(0), B location_coordinate(0,door,X,Y,Z) }
15     <- +going(0), move(X,Y,Z);
```

Listing 3: Partial code of the GWENDOLEN agent.

FOL Predicate	Gwendolen	
$at(x, y) \wedge wayP(x, y) = i$	$\mathcal{B}_{\texttt{jackal}}\, going(i)$ $\mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3)$	$\wedge$
$inspected(i)$	$\mathcal{B}_{\texttt{jackal}}\, inspected(i)$	
$radiationStatus \notin \{red, orange\}$	$\neg\mathcal{B}_{\texttt{jackal}}\, danger_red$ $\neg\mathcal{B}_{\texttt{jackal}}\, danger_orange$	$\wedge$
$radiationStatus \in \{red, orange\}$	$\mathcal{B}_{\texttt{jackal}}\, danger_red$ $\mathcal{B}_{\texttt{jackal}}\, danger_orange$	$\vee$
$\forall\, x, y \in \mathbb{R} \bullet wayP(x, y) \neq i + 1$	$\mathcal{B}_{\texttt{jackal}}\, going(12)$ $\mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3)$	$\wedge$
$wayP(x, y) = i \wedge command = move(x, y)$ $command = inspect(i)$	$\mathcal{B}_{\texttt{jackal}}\, going(i)$ $D_{\texttt{jackal}}\, inspect$	

Table 6: A mapping of RCL contract predicates to the AJPF property specification language

selected for execution. The plan body is executed sequentially, in this case first the bookkeeping belief `going` is added (`+`) and then the action `move` is called with the coordinates of the desired location.

The second plan, `movebase_result(Id, Result)` on line 5, has three variations. All of them are triggered by adding the `movebase_result` belief, which reports that the robot's movement (controlled by the move base library) is complete, with the second parameter indicating either success (value 3) or failure (value 2). The first variation (lines 5–6) is the main plan, which tests where the rover was going before the action succeeded (`going(L1)`) and where the next location is, and then removes the outdated bookkeeping belief, performs the `inspect` action, add a belief that the location $L1$ has been inspected, and then adds the goal to inspect the next location. The second variation (line 7) is for when the robot is moving to the decontamination and should thus not take any additional actions. The third variation (line 9) is for when a `move` action fails, in which case we simply log that it has failed.

Finally, the third (line 12) and fourth plans (line 14) are triggered by the addition of the beliefs `danger_red` or `danger_orange`, respectively. In both cases we test that we are not yet heading to the entrance (`going(0)`) and get its coordinates from the belief base. Then, if the test succeeds, we add the bookkeeping belief and send the `move` action for execution (this is published in a ROS topic that is subscribed to by a move base node).

Clearly the Gwendolen agent uses a different set of abstractions to the RCL contracts. The AJPF property specification language has constructs for referring to the beliefs and actions of an agent – $\mathcal{B}_{ag}\,\phi$ means that ϕ appears in the agent ag's belief base and $D_{ag}\phi$ means that the last action performed by agent ag was ϕ. The expressions, ϕ, map naturally to expressions in the agent program. Table 6 presents a mapping from the properties in the RCL contracts to properties about this Gwendolen program expressed in the AJPF property specification language.

Some of the mappings in Table 6 are straightforward, but a couple are less so. While the Gwendolen code utilises location coordinates, the AJPF property specification language does not handle floats well so this information is abstracted to reasoning at the level of waypoints – for instance the property of being at coordinates (x, y) which are the coordinates of waypoint i (the first property in table 6)

becomes a belief that the Jackal rover was going to waypoint i ($\mathcal{B}_{\texttt{jackal}}\, going(i)$) and that the move action has completed successfully ($\mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3)$ – here we leverage a detail of the AJPF property specification language where we may use _ to stand in for any result, in this case the ID number returned by movebase). Because model-checking is finite state, and we are performing program model checking on a program working with a specific scenario the property that there is no waypoint $i+1$ (property 5 in table 6) becomes the property that the agent is at waypoint 12.

The AJPF property specification language is LTL while the RCL contracts are expressed in FOL. In general the properties in the RCL contracts are of the form $H \Rightarrow C$ where H expresses some condition on the inputs to the node and C expresses some property of the outputs. These become properties of the form $\Box(H' \Rightarrow \Diamond C')$ (where H' is the form of H as mapped into the AJPF property specification language and C' is the form of C). Since we have no quantifiers in LTL, and model checking is finite state, a general property about *all waypoints* in RCL becomes n properties about each of the n waypoints for AJPF.

In all there are three guarantees specified in the RCL contract for this node. We consider each of these in turn.

Guarantee 1 (lines 15–18 in Listing 2) This guarantee states that "If the Robot is at, and has inspected waypoint i, and the radiation level was not dangerous, then it will move to the next waypoint or to waypoint 0 (the exit) if there is no next waypoint. "

We express this in the AJPF property specification language as 12 properties of the form:

$$\begin{aligned}\varphi_1^i = \Box((&\mathcal{B}_{\texttt{jackal}}\, going(i) \wedge \mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3) \wedge \\ &\neg\, \mathcal{B}_{\texttt{jackal}}\, danger_red \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, danger_orange \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, going(0)) \\ &\Rightarrow \Diamond B_{\texttt{jackal}}\, going(i+1))\end{aligned}$$

where i is instantiated with the waypoint numbers from 0 to 11 – e.g., as

$$\begin{aligned}\varphi_1^4 = \Box((&\mathcal{B}_{\texttt{jackal}}\, going(4) \wedge \mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3) \wedge \\ &\neg\, \mathcal{B}_{\texttt{jackal}}\, danger_red \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, danger_orange \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, going(0)) \\ &\Rightarrow \Diamond B_{\texttt{jackal}}\, going(5))\end{aligned}$$

Plus a 12th property:

$$\begin{aligned}\varphi_1^1 2 = \Box((&\mathcal{B}_{\texttt{jackal}}\, going(12) \wedge \mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3) \\ &\Rightarrow \Diamond B_{\texttt{jackal}}\, going(0))\end{aligned}$$

The RCL contract for this guarantee has an additional condition $inspected(i) = TRUE$ which we have omitted giving us a more general property. We have also omitted the conditions that the radiation status is not orange or red from $\varphi_1^1 2$ again to give us a more general property (in this case it remains true no matter what the radiation status since the robot also goes to waypoint 0 in the case of dangerous radiation).

Guarantee 2 (Lines 20–21 in Listing 2) This guarantee states that "if the Robot is at, and has not inspected, waypoint i, then inspect it." We verified two variants of this guarantee in AJPF. The first, φ_2^i, are n properties (with i instantiated from 1 to 12) that follow directly the mapping of the RCL contract predicates into the AJPF property specification language as presented in Table 6.

$$\begin{aligned}\varphi_2^i = \Box((&\mathcal{B}_{\texttt{jackal}}\, going(i) \wedge \mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3) \wedge \\ &\neg\, \mathcal{B}_{\texttt{jackal}}\, danger_red \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, danger_orange \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, going(0)) \\ &\Rightarrow \Diamond D_{\texttt{jackal}}\, inspect)\end{aligned}$$

However, since the *inspect* action doesn't actually specify the waypoint inspected, it was possible for the Gwendolen program to satisfy this property while missing a waypoint (e.g., it could fail to inspect waypoint 4, but the property would become true when waypoint 5 was inspected). Therefore we also verified a variant that checked that the agent *believed* it had checked all waypoints - utilising the fact that the agent adds this belief after each waypoint is inspected. Our variant $\varphi_2^{i'}$ properties were:

$$\begin{aligned}\varphi_2^{i'} = \Box((&\mathcal{B}_{\texttt{jackal}}\, going(i) \wedge \mathcal{B}_{\texttt{jackal}}\, movebase_result(_, 3) \wedge \\ &\neg\, \mathcal{B}_{\texttt{jackal}}\, danger_red \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, danger_orange \wedge \neg\, \mathcal{B}_{\texttt{jackal}}\, going(0)) \\ &\Rightarrow \Diamond \mathcal{B}_{\texttt{jackal}}\, inspect)\end{aligned}$$

```
1 input := radiation_at(i)
2 IF (input < 120) THEN
3     output := green
4 ELSE
5     IF (input < 250) THEN
6         output := orange
7     ELSE
8         output := red
```

Listing 4: Program for the radiation sensor node.

While these properties are not such a close mapping from the RCL contract, we believe they better capture the intent behind the guarantee.

Guarantee 3 (Lines 23–25 in Listing 2) This guarantee states that "if the Robot inspects a waypoint and the radiation level is dangerous, or there are no more waypoints, then return to the original waypoint (the exit)". This became two properties (one for a radiation status of red and one for orange). The final part of the guarantee (about the behaviour when there are no more waypoints, was effectively covered by φ_1^12)

$$\varphi_3^a = \Box(\mathcal{B}_{\texttt{jackal}}\, \mathit{danger_red} \Rightarrow \Diamond\mathcal{B}_{\texttt{jackal}}\, \mathit{going}(0))$$

$$\varphi_3^b = \Box(\mathcal{B}_{\texttt{jackal}}\, \mathit{danger_orange} \Rightarrow \Diamond\mathcal{B}_{\texttt{jackal}}\, \mathit{going}(0))$$

These LTL properties were only used for verification of the agent node at design time, using the AJPF model checker and were not used elsewhere (e.g., for the runtime monitors). A discussion of AJPF and its efficiency can be found in [72].

4.4.4 Radiation Sensor Node

The **Radiation Sensor** node interprets information from the (simulated) radiation sensor, categorising the values into either `green` (low), `orange` (medium), or `red` (high). We modelled the behaviour of the **Radiation Sensor** as a simple program in a Hoare Logic-style language [73] and proved properties corresponding to the node's guarantee by hand.

The **Radiation Sensor** node's contract assumes that the measured radiation is 0 or positive. Its guarantee specifies that if the node is asked to inspect waypoint i (`command = inspect(i)`) then:

1. the proposition *inspected*(i) becomes true, *inspected*(i) = *TRUE*;
2. low-level radiation is categorised as `green`, $0 <= r < 120 \Rightarrow$ *radiationStatus* = *green*;
3. medium-level radiation is categorised as `orange`, $120 <= r < 250 \Rightarrow$ *radiationStatus* = *orange*; and,
4. high-level radiation is categorised as `red`, $250 <= r \Rightarrow$ *radiationStatus* = *red*.

We proved these four properties by hand using Hoare Logic. If `P` is our program from Listing 4 then:

1. $\{\top\}$ `P`$\{$*input* = *radiation_at*(i)$\}$ – The input to the node is always the radiation level at i. This corresponds to the requirement that *inspected*(i) = *TRUE* in the contract.
2. $\{$*radiation_at*(i) $< 120\}$ `P`$\{$*output* = *green*$\}$ – if the radiation level at i is less than 120 then, after execution of the program, *output* is green.
3. $\{120 \leq$ *radiation_at*(i) $< 250\}$ `P`$\{$*output* = *orange*$\}$ – if the radiation level at i is between 120 and 250 then, after execution of the program, *output* is orange.
4. $\{$*radiation_at*(i) $\geq 250\}$ `P`$\{$*output* = *red*$\}$ – if the radiation level at i is greater than 250 then, after execution of the program, *output* is red.

```
1  {radiation_at(i) < 120}
2  input := radiation_at(i)
3  {input < 120} − Assignment Axiom
4   IF (input < 120) THEN
5      {input < 120& input < 120}
6      {input < 120 & green = green} − strengthening
7       output := green
8      {input < 120 & output = green} − Assignment Axiom
9      {output = green} − weakening
10  ELSE
11     {input < 120 & input >= 120}
12     {FALSE} − strengthening
13  IF (input < 250) THEN
14         {FALSE & input < 250}
15         {FALSE} − strengthening
16          output := orange
17         {FALSE} − Assignment axiom
18        ELSE
19           {FALSE & input >= 250}
20           {FALSE} − strengthening
21            output := red
22           {FALSE} − Assignment axiom
23     {FALSE} − Conditional Rule
24     {output = green} − weakening
25 {output = green} − Conditional Rule
```

Listing 5: Hoare Logic Proof that $\{radiation_at(i) < 120\}\mathbf{P}\{output = green\}$. The program is denoted by **bold** text, and the proof steps are indicated by $\{braces\}$.

```
1  node agent{
2  ...
3  topics(
4  gazebo_radiation_plugin/Snapshot command matches(out.command),
5  gazebo_radiation_plugin/Snapshot inspected matches(in.inspected),
6  gazebo_radiation_plugin/Snapshot at matches(in.at),
7  gazebo_radiation_plugin/Snapshot wayPNow matches(in.wayP),
8  gazebo_radiation_plugin/Snapshot radiationStatus matches(in.radiationStatus)
9  )
10 ...
11 guarantee(forall(x', y' in REAL, i in NATURAL | in.at(x', y') == TRUE and in.wayP(x', y') == i and in
       .inspected(i) == FALSE -> out.command == inspect(i) ))
12 ...
13 }
```

Listing 6: A snippet of the **Agent**'s contract from Listing 2.

We show the proof for property (2) in Fig. 5. The other proofs follow a similar pattern, and can be found in B.

4.5 Step 4b: Automatic Synthesis of Runtime Monitors

This step takes the RCL contracts from Step 2 (§4.2) and synthesises RML monitors that are compatible with the ROSMonitoring framework [57]. As mentioned in §3.3, our tool VANDA parses the RCL contracts, and then automatically generates the monitors and ROSMonitoring configuration files. VANDA uses a *contract translator* to produce the configuration file and structure of each monitor, and calls a *FOL translator* to translate each guarantee.

Listing 7 shows the RML that was derived from the RCL contract in Listing 2. Lines 1–6 contain the event types corresponding to the contract's topics, while on lines 8–12 we have the RML terms for the contract's guarantees. These terms are obtained through the process shown in §3.6. RML offers a Prolog-like notation, where variables that are not of interest can be replaced with the _ symbol. This is interpreted by RML as a wildcard variable, which can be assigned to any value with no restrictions.

To aid the reader's understanding of this translation step, we show in more detail how a specific part of this RML specification has been generated. The approach is described in § 3.6, and the same reasoning is applied to the translation of the other contracts.

Listing 6 shows a snippet of Listing 2 that contains only one of the **Agent** node's guarantees. The `guarantee` in Listing 6 is translated into the RML term `t2` in Listing 7 using the (*guarantee*) rule

```
1  at(x, y) matches { topic: 'gazebo_radiation_plugin/Snapshot', at: {posX: x, posY:y} };
2  wayP(x, y, i) matches { topic: 'gazebo_radiation_plugin/Snapshot', wayPNow: {posX: x, posY: y, id:
       i} };
3  radiationStatus(Lvl) matches { topic: 'gazebo_radiation_plugin/Snapshot', radiationStatus: { level:
       Lvl } };
4  command(Cmd, x, y) matches { topic:'gazebo_radiation_plugin/Snapshot', command: {command: Cmd, posX:
       x, posY: y } };
5  command(Cmd, i) matches { topic:'gazebo_radiation_plugin/Snapshot', command: {command: Cmd, id: i} };
6  inspected(i) matches { topic: 'gazebo_radiation_plugin/Snapshot', inspected{ id: i } };
7
8  t1 = {let x1, y1, i; ((¬ at(x1,y1) ∨ ¬ wayP(x1, y1, _) ∨ ¬ radiationStatus(green))) ∨ {let x2,y2;
       (wayP(x2, y2, i+1) ∧ command(move, x2, y2))}}*;
9
10 t2 = {let x, y, i; (¬ at(x, y) ∨ ¬ wayP(x, y, i) ∨ inspected(i)) ∨ (command(inspect, i))}*;
11
12 t3 = {(radiationStatus(green) ∨ {let x, y; wayP(x, y, 0) ∧ command(move, x, y))})}*;
```

Listing 7: The **Agent**'s RML derived from the RCL contract shown in Listing 2.

```
1  monitors:
2  - monitor:
3      id: monitor_agent
4      log: ./agent_log.txt
5      topics:
6       - {action: log, name: gazebo_radiation_plugin/Snapshot, type:
       gazebo_radiation_plugins.msg.Snapshot}
```

Listing 8: The monitor configuration file for the **Agent** node.

in Figure 4.

The body of the `guarantee` consists of a universal quantifier (`forall`) that contains an implication (`->`). First, we use the (*implies*) rule in Figure 4 to translate the implication; (*implies*) negates the left operand and puts it in disjunction with the right operand. The left operand is: `in.at(x', y') == TRUE and in.wayP(x', y') == i and in.inspected(i) == FALSE`, which produces the following RML specification:

`(¬ at(x, y) ∨ ¬ wayP(x, y, i) ∨ inspected(i))` .

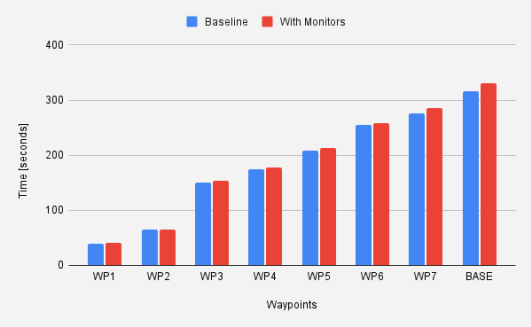
Note, that the RML is negated because `(a -> b) = (¬ a or b)`. Then, the right operand `in.command==inspect(i)` is directly mapped into RML as: `command(inspect, i)`. This follows from the (*equals*) rule in Figure 4, which maps the command into its corresponding event type and adds it to the set of event types.

Next, the variables derived from the universal quantifier are added to the RML specification. Thus, we obtain `{let x, y, i; t}`, where `t` is the previously created RML specification (the one denoting the implication). Through this quantification, in RML we can generalise `t` over the set of variables `{ x,y,i }`. Finally, since the `guarantee` ranges over all possible instantiations for the variables, an `*` is added at the end of RML specification (as specified in the (*guarantee*) rule in Fig. 4). This operator, as in regular expressions, requires the RML specification to be matched multiple times. As previously mentioned, the same reasoning is applied to the other guarantees in Listing 2.

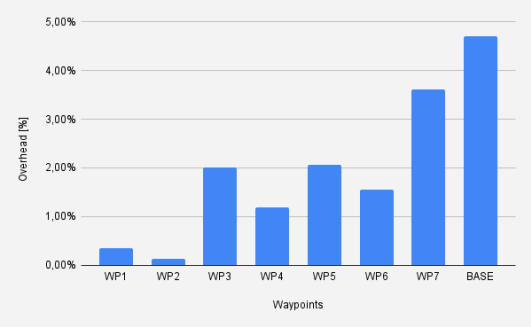
Once the RML specification (Listing 7) and the configuration file (Listing 8) have been generated, the RV step can be applied, as previously presented in Fig. 3. Hence, the RML specification can be used to synthesise an oracle, which will then be queried at runtime by the corresponding ROS monitors (automatically synthesised from configuration files).

ROSMonitoring uses the configuration file, shown in Listing 8, to generate the **Agent** node's monitor. Lines 3 & 4 show the agent monitor's identifier and the location where its log files should be stored. The monitor also requires the list of topics to subscribe to (lines 5–6) to observe the events at runtime. These topics are obtained from lines 7–12 in Listing 2. Note that, in this specific scenario, we only have one single topic, called `Snapshot`, that contains the information we need.

As previously mentioned, ROSMonitoring automatically synthesises runtime monitors as additional nodes in the ROS program. The monitor nodes that ROSMonitoring synthesises subscribe to the topics of interest listed in the configuration file (derived from the RCL specification). After that, they collect the messages that are published on those topics and perform the runtime analysis of the RML specification. If a monitor observes a violation of an RML specification, it reports the violation to the entire system by publishing a specific error message. This information can be used by the system to



(a) The y-axis indicates the time at which the rover completes its visit.



(b) The y-axis indicates the percentage increase with respect to the baseline.

Figure 8: Overhead experiment results in the remote inspection case study. The x-axis represents the visited waypoints. BASE is WP0, the door location. When reaching WP7 the rover detects dangerous radiation and then decides to move to the door for decontamination.

promptly react and possibly recover from the erroneous behaviour.

An additional step of instrumenting the implementation of the case study was necessary to fully capture the execution traces that the ROSMonitoring monitors needed to observe. For example, this instrumentation step included publishing information that was kept within the **Agent**, such as beliefs about which waypoint the robot is at, to a ROS topic. This allowed the ROSMonitoring monitors to observe this information and give their verdict. Note that the instrumentation step is only necessary to track the behaviour of components, such as the **Agent**, which otherwise would not be observed; it does not change the behaviour of the system components.

Using RML monitors for RV of our contracts only causes slight overheads. We conducted an analysis of the overhead introduced by the presence of monitors in our case study. Following a similar approach to [57], first we timed our system patrolling its simulated environment without monitors to establish a baseline, then timed the system with the monitors. Fig. 8a shows the time required for the robot to visit all the waypoints and return to the exit, averaged over 30 runs. The baseline, without monitors, is shown in blue, while the scenario with ROSMonitoring's monitors performing runtime verification is shown in red. As it can be observed, the overhead introduced by the monitors is negligible. Fig. 8b provides a more detailed view, showing the percentage overhead at waypoint (relative to the baseline execution without the monitors). Further experiments on the overhead of ROSMonitoring are presented in [57], where the overhead of RML monitors in ROSMonitoring was analysed. This analysis empirically demonstrated that RML monitors were introducing almost no overhead to the robotic system, as long as ROSMonitoring was used to detect failures and not to enforce correctness.

Finally, thanks to the structure of the RML terms that we synthesise, the RV monitors operate in linear time (with respect to the length of the analysed trace). For more information about the time complexity of RML monitoring, the reader can find additional details in [59].

5 Beyond the Case Study

This section goes beyond our case study, to demonstrate other features of our framework, In §5.1, we provide a worked example of how our framework enables the implementation of one node to be swapped with another, as long as the replacement node satisfies the specification. §5.2 explores how our approach to composing the node specifications can be applied to systems that are not implemented in ROS. But first, we discuss the general utility of our approach, beyond what is presented in §4. The analysis of our approaches wider applicability, and the two formal models in this chapter are available from the Zenodo repository that accompanies this paper: `https://doi.org/10.5281/zenodo.6941344`.

While we have shown the applicability of the approach to our case study, there remains a question of whether it is useful in general for the verification of robotic systems. As a step towards assessing this, we examined fifty-six papers containing verifications of autonomous systems identified in our previous survey paper [10]. For each paper, we assessed if our approach would be applicable to the

system it describes. We identified 20 papers with a case study to which our general approach would be applicable. Three of these, specifically involved the verification of ROS nodes. Of these 20 papers, nine involved the verification of a component within a larger autonomous system and 10 involved the system-level verification of a modular system (the remaining one applied runtime verification to a modular system).

Of the systems where we deemed our approach to be inapplicable, the majority (20 in total) involved the verification of distributed or swarm systems where an unspecified number of similar components act to produce system level behaviour and, at present, our formal system is not equipped to work with such architectures. Four examples involved the verification of monolithic systems. In general, the behaviour of these monolithic systems was comparatively simple when compared to modular systems (typically these were hybrid control systems where the verification focused on obstacle avoidance behaviour). We identified nine verification attempts as borderline. These involved fixed-size teams of independent systems – often individual robots but sometimes independent, communicating systems within a smart home or similar environment. Our approach should be extensible to such systems by viewing sub-systems as components in a hierarchical fashion. One paper contained an analysis of a failed verification attempt – we also classified this as borderline.

5.1 Compositionality enables Interchangeability

One of the advantages of our approach is that we can swap a node's implementation with another (similar) implementation, which may be modelled and verified using different specification languages and verification tools to the original node, as long as the new implementation conforms to the FOL contract. For example, instead of using a Gwendolen BDI agent (as we do in §4) we might choose a simpler way of making the systems' executive decisions that still satisfies $\mathcal{G}_A(\overline{o_A})$.

To demonstrate the modularity of our approach, this section describes a re-specification and verification of the **Agent** (§4.4.3) node in Dafny [74]. That is, we replace the Gwendolen BDI implementation and verification with this new Dafny implementation. Dafny is a programming language enriched with specification constructs – for example: pre-/post-conditions and loop invariants[13]. This enables the functional correctness of Dafny programs to be statically verified, by translating them into the intermediate verification language Boogie [75] and using the theorem prover Z3 [76] to automatically discharge the proof obligations for the specification statements in the program.

The idea here is that instead of using a Gwendolen BDI program, the decision-making algorithm(s) will be implemented in a general-purpose programming language, for example C++ or Python for compatibility with ROS. So we use Dafny to implement and verify the algorithms that the new decision-making component will execute. Our prior work demonstrates that the correspondence between Dafny and general-purpose languages, such as Python, makes it relatively straightforward to translate between formal models and implemented code [43].

Listing 9 shows our Dafny model, in which the `Agent` method (line 5) provides an alternative implementation of the agent's behaviour to the Gwendolen BDI version that was presented in §4.4.3. Importantly, the Dafny implementation follows the specification in its contract (Table 4).

In Dafny, pre-conditions (assumptions) are indicated by the `requires` keyword. Lines 6–9 of Listing 9 show the `Agent` method's pre-conditions, taken from its contract. One small change is that where the **Agent**'s contract has an assumption that the radiation status will be either red, orange, or green; our Dafny version encodes this constraint as the `RadiationLevel` datatype (line 3) using the radiation values, and using this in the parameters (for `radstat`) of the `Agent` method. Importantly, if the `Agent` method is called by another method, then the calling method's guarantees must not violate the assumptions of the `Agent` method.

Post-conditions (guarantees) in Dafny are indicated by the `ensures` keyword. Lines 10–13 of Listing 9 show the `Agent` method's post-conditions, which are more detailed than the **Agent**'s contract, because they also check the status of the software controlling the rover's wheels before instructing it to carry out any commands. The post-condition on line 10 specifies that no actions are assigned when the wheels are not ready. The post-conditions on lines 11–13 correspond to the **Agent**'s guarantees from its contract (Table 4). For verification in Dafny, we must include loop invariants (lines 21–25), which help the verification tool (Z3) to prove the post-conditions. We automatically discharged the

[13]Note that our Dafny model uses `function method` constructs from Dafny 3.x which are deprecated in the latest version (4.9.1). However, the proof obligations are still discharged using normal functions in Dafny version 4.9.1.

```
1  datatype Action = Inspect | MoveNext
2  datatype Command = Command(a:Action, w:int)
3  datatype RadiationLevel = red | orange | green
4
5  method Agent(waypoints: seq<int>, currentpos:int, radstat:RadiationLevel, wheelsready:bool) returns
       (actions: seq<Command>)
6  requires currentpos ⩾ 0 ∧ currentpos ⩽ 12; //13 waypoints to patrol
7  requires |waypoints| = 13;
8  requires ∀ i:int · 0⩽i<|waypoints| ⇒ waypoints[i] = i;
9  requires ∀ i:int · 0⩽i<|waypoints| ⇒ i in waypoints;
10 ensures (wheelsready) = false ⇒ actions =[];//safety check: if the hardware is not ready then do
       nothing.
11 ensures ∀ i:int · 0⩽i<|waypoints| ⇒ (wheelsready ∧ at(i) ∧ inspected(i) ∧ radstat ≠ red ∧ radstat
       ≠ orange ⇒ ∃ next:int · 0⩽next<|waypoints| ∧ Command(MoveNext, next) in actions);
12 ensures ∀ i:int · 0⩽i<|waypoints| ⇒ (wheelsready ∧ at(i) ∧ (! inspected(i)) ⇒ Command(Inspect,
       i) in actions);
13 ensures wheelsready ∧ (radstat = red ∨ radstat = orange) ⇒ Command(MoveNext, 0) in actions;
14 {
15     var time, next := 0, 0;
16     var current:=currentpos;
17     actions := [];
18
19     if(wheelsready){
20         while(wheelsready ∧ time <200)
21         invariant next in waypoints;
22         invariant current in waypoints;
23         invariant time > 0 ∧ (radstat = red ∨ radstat = orange) ⇒ Command(MoveNext, 0) in actions ∧
       next = 0;
24         invariant time > 0 ∧ at(current) ∧ (! inspected(current)) ⇒ Command(Inspect, current) in
       actions;
25         invariant time > 0 ∧ at(old(current)) ∧ inspected(old(current)) ∧ radstat ≠ red ∧ radstat ≠
       orange ⇒ Command(MoveNext, next) in actions ∧ current = old(next);
26         {
27             if(at(current) ∧ inspected(current) ∧ radstat ≠ red ∧ radstat ≠ orange){
28                 next := getnextwaypoint(current);
29                 actions := actions + [Command(MoveNext, next)];
30             }
31             if(at(current) ∧ (! inspected(current))){
32                 actions := actions + [Command(Inspect, current)];
33             }
34             if(radstat = red ∨ radstat = orange){
35                 next := 0;
36                 actions := actions + [Command(MoveNext, next)];
37             }
38             time := time +20;
39             current := next;
40         }
41     }
42 }
```

Listing 9: Dafny Agent

associated proofs in Dafny 3.x with Z3 using Visual Studio Code on Ubuntu.

Another difference between our Dafny and Gwendolen implementations is that we had to add a notion of `time` to the Dafny model (the `time` variable is declared on line 15 and updated on line 38) to be able to prove loop termination. Dafny is primarily concerned with program safety, so termination is necessary for complete verification of Dafny programs.

This Dafny program verifies the decision-making algorithm, using a language that is closer to those in which ROS nodes can be implemented. Once the Dafny program has been verified against the **Agent** contract, it can be carefully reimplemented as a ROS node. The Dafny **Agent** can also be used with our calculus (§3.4) because it preserves the contract's assumptions and guarantees. Although the RCL and Dafny contracts use slightly different datatypes, we can see the correspondence between them easily. Specifically, the Dafny model returns a sequence of (action, waypoint) pairs that represents the `CommandSet` of the RCL contract which contains move and inspect commands. The `CommandSet` can thus be reconstructed from the Dafny output and vice versa if needed. Similarly, the belief base of the Gwendolen agent can be checked and shown to contain the same information as the `CommandSet`.

Not only does this allow the user to develop multiple models (and potentially implementations) of the same node, it also facilitates the use of predefined or library functions that have been verified previously and also meet the required contracts. This can potentially streamline the verification step by supporting the use of previously verified robotic system nodes.

The Dafny and Gwendolen agents described in this work provide potential alternatives to illustrate our approach. A developer should choose the formalism that most suitably matches their needs. Users of our framework should ensure that their FOL contracts are captured correctly in the formalism that they choose. To support this, systematic mappings between FOL and other logics/formalisms can be

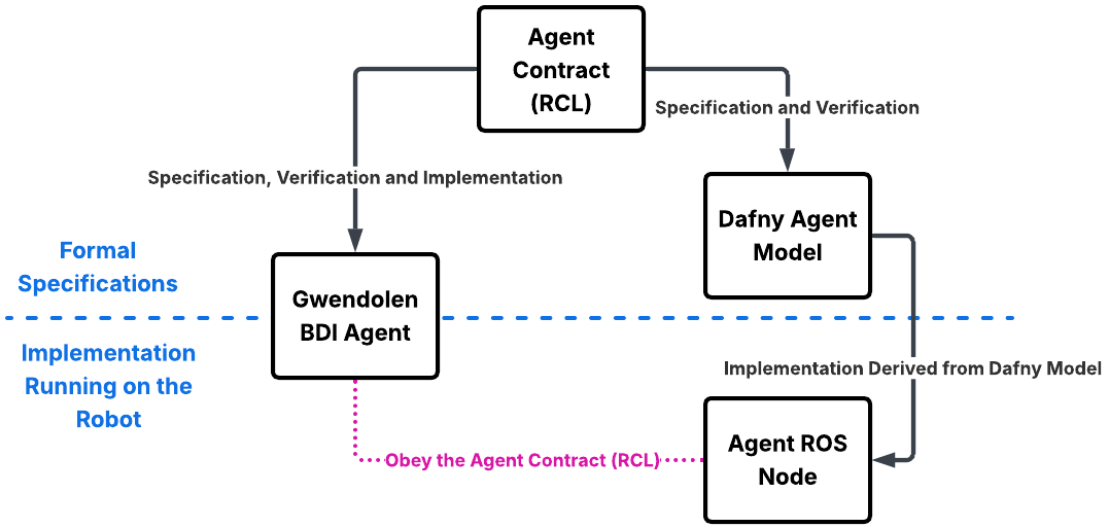


Figure 9: Summary of the different agent models, showing the differences and similarities. Both the Gwendolen BDI and Dafny agent models are specified by the RCL contract, but the Dafny model must be implemented in a ROS node to be used as the **Agent**.

used but such mappings are out of scope for this paper.

Both the Dafny and Gwendolen agents are designed to visit a specific number of waypoints. This is necessary in Gwendolen since model-checking is used for verification. However, our Dafny model could be more liberal here and, since all sequences in Dafny are finite, omit the specific sequence size. The main modification would be to add a precondition: `requires currentpos in waypoints` to make sure that the input is valid. The `getNextWaypoint` function must also consider the size of the waypoints sequence when producing the next waypoint. In comparison with Dafny, Gwendolen supports temporal logic specifications which may be useful for a developer wanting to also verify additional temporal properties about their agent. Both Dafny and Gwendolen can be executed but Gwendolen, via ROSbridge, can be used directly on the robot. No such support currently exists for Dafny. Both Dafny and Gwendolen allow users to specify properties that were not specified in our FOL contracts including, for example, frame conditions in Dafny and temporal logic properties in Gwendolen. Dafny's verifier is a first-order SMT solver which makes it natural to express FOL contracts in Dafny.

For simplicity, the Dafny and Gwendolen agents that we present in this paper are very similar in terms of functionality but, in practice, they could be slightly different so long as they obey the same FOL contract. For example, one of the agents could incorporate other behaviours than just inspection. It could both inspect and take samples from the locations that it visits. It may even stop to do self-maintenance before proceeding to the next waypoint.

We illustrate the relationships between the different agent models in Fig. 9. Here, we can see that the RCL contract is a specification for both the Dafny program and Gwendolen BDI agents, and that they are verified using formal techniques (model-checking for the Gwendolen BDI agent and Theorem Proving for the Dafny agent model). However, one of the main differences between these two representations is that the Gwendolen BDI agent is both a formal specification and an implementation that can be used directly in ROS, whereas the Dafny agent model needs to be manually implemented in a ROS node. Providing that the ROS implementation of the Dafny agent model is correct, both the Dafny and Gwendolen BDI agents obey the original RCL contract. Though their actual behaviours may differ as described above, depending on the choice and needs of the developer.

5.2 Towards Applying our Approach to Non-ROS Systems

So far, we have focussed on ROS systems; this section explores how our approach can be adapted to robotic systems that do not use ROS. We use previous work [28] as our example, in which heterogeneous verification methods are used to verify an autonomous rover that is modelled in Simulink. The rover's mission is to autonomously navigate around a grid-world of known size, visiting waypoints to collect data, recharging as necessary. This example has similar functionality to our case study system in §4, but does not use ROS.

Fig. 10. shows the rover's architecture in Architecture Analysis and Design Language (AADL). This rover contains multiple connected modules, some of which provide functionality that is more critical than others – for example the **ReasoningAgent**, which is the core decision-making component of the system. To account for this mixed-criticality, the work in [28] begins by eliciting the system's requirements using an approach driven by a detailed hazard analysis.

One of the system's most important requirements is that *the rover shall not run out of battery.* This particular requirement can be viewed as a system-level contract that relies on the behaviour of multiple system components. For example, the **Battery_Interface** must function correctly, ensuring that the rover's goal location is set to the charge station when needed; and the **ReasoningAgent** must correctly select the shortest path, to conserve battery power.

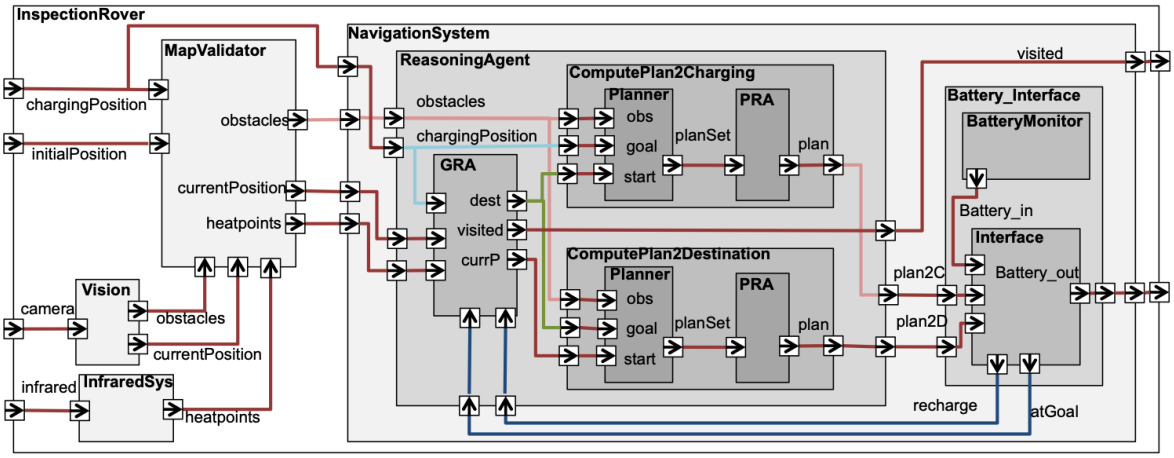


Figure 10: Architecture of autonomous rover from [28].

The work in [28] uses CoCoSim to enable the compositional verification of the Simulink model of the system. This approach works by attaching contracts to nodes in the system and then defining a top node where the system-level contracts are specified. Here, compositional verification involves verifying (by model-checking) that the node-level contracts imply the system-level contracts.

A key point of our approach is that it can derive system-level properties from the node-level contracts. In [28], both component-level contracts and system-level contracts are derived from the requirements and the component-level contracts are attached to individual system components. CoCoSim was then used to verify that the system-level properties could be deduced from the component-level contracts. The difference between this approach and ours is that CoCoSim requires a system-level property against which to verify systems, whereas we support deriving properties about a system's behaviour for systems built from (potentially independently developed) components.

To compare with this work and investigate how our approach performs on this non-ROS system, we examine its component-level contracts to see if we can derive similar system-level properties to those used in [28]. For this example, we focus our attention specifically on the **NavigationSystem** shown in Fig. 10, which is composed of a **ReasoningAgent** and a **Battery_Interface**. The **ReasoningAgent** contains a Goal Reasoning Agent (**GRA** in Fig. 10), to select the rover's goal location; the **ComputePlan2Charging** component, which generates a plan to the charging point; and the **ComputePlan2Destination** component, which generates a plan to other destinations. The **Battery_Interface** contains a **BatteryMonitor** component, which regularly checks the battery level; and the **Interface**, which updates the **GRA** when the rover needs to recharge.

Despite this system not being implemented in ROS, we are able to apply our calculus to it because

it is built from components that act similarly to a ROS nodes. We take Fig. 10 as our system model (which maps to Step 1 of our approach in §3) and apply our calculus to the **ReasoningAgent** and **Battery_Interface**.

First, we apply the calculus to the subcomponents of the **ReasoningAgent**. Because the output of the **GRA** is the input of both **ComputePlan2Charging** and the **ComputePlan2Destination**, we use the branching output rule, **R2**:

$$\frac{\begin{array}{l}\forall \overline{i_{GRA}}, \overline{o_{GRA}} \cdot \mathcal{A}_{GRA}(\overline{i_{GRA}}) \;\Rightarrow\; \Diamond \mathcal{G}_{GRA}(\overline{o_{GRA}}) \\ \forall \overline{i_{CPC}}, \overline{o_{CPC}} \cdot \mathcal{A}_{CPC}(\overline{i_{CPC}}) \;\Rightarrow\; \Diamond \mathcal{G}_{CPC}(\overline{o_{CPC}}) \\ \forall \overline{i_{CPD}}, \overline{o_{CPD}} \cdot \mathcal{A}_{CPD}(\overline{i_{CPD}}) \;\Rightarrow\; \Diamond \mathcal{G}_{CPD}(\overline{o_{CPD}}) \\ \overline{o_{GRA}} = \overline{i_{CPC}} \cup \overline{i_{CPD}} \\ \vdash \left(\begin{array}{l} \forall \overline{o_{GRA}}, \overline{i_{CPC}} \cdot \mathcal{G}_{GRA}(\overline{o_{GRA}}) \;\Rightarrow\; \mathcal{A}_{CPC}(\overline{i_{CPC}}) \wedge \\ \forall \overline{o_{GRA}}, \overline{i_{CPD}} \cdot \mathcal{G}_{GRA}(\overline{o_{GRA}}) \;\Rightarrow\; \mathcal{A}_{CPD}(\overline{i_{CPD}}) \end{array} \right)\end{array}}{\left(\forall \overline{i_{GRA}}, \overline{o_{CPC}}, \overline{o_{CPD}} \cdot \mathcal{A}_{GRA}(\overline{i_{GRA}}) \;\Rightarrow\; \Diamond \mathcal{G}_{CPC}(\overline{o_{CPC}}) \wedge \Diamond \mathcal{G}_{CPD}(\overline{o_{CPD}}) \right)}$$

Intuitively this means that, when the assumptions of the **GRA** component hold, then eventually the guarantees of the **ComputePlan2Charging** (CPC) and the **ComputePlan2Destination** (CPD) hold. This also captures the requirement for this system that all plans are valid and free from obstacles [28].

Similarly we can use the union of inputs rule, **R3**, to link the three components that provide input to the **Interface** (I): the **ComputePlan2Charging** (CPC), the **ComputePlan2Destination** (CPD), and the **BatteryMonitor** (BM).

Using **R3**, we can derive the following:

$$\frac{\begin{array}{l}\forall \overline{i_{CPC}}, \overline{o_{CPC}} \cdot \mathcal{A}_{CPC}(\overline{i_{CPC}}) \;\Rightarrow\; \Diamond \mathcal{G}_{CPC}(\overline{o_{CPC}}) \\ \forall \overline{i_{CPD}}, \overline{o_{CPD}} \cdot \mathcal{A}_{CPD}(\overline{i_{CPD}}) \;\Rightarrow\; \Diamond \mathcal{G}_{CPD}(\overline{o_{CPD}}) \\ \forall \overline{i_{BM}}, \overline{o_{BM}} \cdot \mathcal{A}_{BM}(\overline{i_{BM}}) \;\Rightarrow\; \Diamond \mathcal{G}_{BM}(\overline{o_{BM}}) \\ \forall \overline{i_I}, \overline{o_I} \cdot \mathcal{A}_I(\overline{i_I}) \;\Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I}) \\ \overline{i_I} = \overline{o_{CPC}} \cup \overline{o_{CPD}} \cup \overline{o_{BM}} \\ \vdash\; \forall \overline{i_I}, \overline{o_{CPC}}, \overline{o_{CPD}}, \overline{o_{BM}} \cdot \mathcal{G}_{CPC}(\overline{o_{CPC}}) \wedge \mathcal{G}_{CPD}(\overline{o_{CPD}}) \wedge \mathcal{G}_{BM}(\overline{o_{BM}}) \Rightarrow\; \mathcal{A}_I(\overline{i_I})\end{array}}{\left(\forall \overline{i_{CPC}}, \overline{i_{CPD}}, \overline{i_{BM}}, \overline{o_I}, \cdot \mathcal{A}_{CPC}(\overline{i_{CPC}}) \wedge \mathcal{A}_{CPD}(\overline{i_{CPD}}) \wedge \mathcal{A}_{BM}(\overline{i_{BM}}) \Rightarrow\; \Diamond \mathcal{G}_I(\overline{o_I}) \right)}$$

Thus we can deduce that when the plan-computing components (**ComputePlan2Charging** and **ComputePlan2Destination**) and the **BatteryMonitor** are functioning correctly, then eventually the **Interface**'s guarantee is preserved. This means that battery usage is computed correctly and that the rover stays in the charger location until it has fully recharged.

Both, this previous approach and the derivations above are able to derive that the rover produces obstacle-free plans. The approach in [28] was able to verify that the rover never runs out of battery but our rules above only allowed us to show that the rover recharges fully when required. These properties are fairly closely related. The reason that we could deduce the property that the rover never runs out of battery was down to the way that the approach using CoCoSim works. There, we had to specify top-level requirements including that the rover never runs out of battery and essentially show that the component-level contracts imply this. Our current approach is more flexible and does not require system-level properties in the same way, rather it derives the properties for the system from the component-level contracts. This has benefits, especially in the domain of autonomous systems where there might be, previously unknown, emergent properties.

This subsection has illustrated how our contract-based compositional verification approach can be applied in the case of non-ROS systems to derive system-level contracts. We used the rover's AADL model (Fig. 10) as our system model, which maps to Step 1 in our approach (§3). Here, we did not need to abstract the model to make it more manageable, but this may be needed for more complicated models. Then we use FOL to write and reason about contracts, which maps to Steps 2 and 3 of our approach. Heterogeneous verification (mapping to Step 4a) is still possible here, guided by the FOL contracts. Automatically generating runtime monitors (Step 4b) is not possible here, because that part of our approach relies on information in the RCL contracts that is specific to ROS. However, generating runtime monitors in a suitable generic framework that can be applicable to non-ROS systems is a useful avenue of future work.

6 Conclusions and Future Work

This paper contributes a compositional approach to the development of verifiable modular robotic systems, which focusses on systems that use the Robot Operating System (ROS) – though parts of it are applicable to non-ROS systems. Each module (or ROS node) is specified using an assume–guarantee contract written in First-Order Logic (FOL), that guide its verification. We also present a calculus for composing these contracts, which caters for sequences, joins, and branches in the system's architecture. Each node can be verified using the most suitable method; some may be amenable to formal verification, while others may not. The verification is driven by the contracts, which specify the minimal set of properties that the verification must be able to show hold for that node (to an appropriate level of confidence for the node, and the system's regulatory environment).

As a safety net, we automatically synthesise formal monitors to verify the contracts' guarantee(s) at runtime. The runtime verification is handled by ROSMonitoring [57], an existing tool for runtime verification of ROS systems. Supporting this approach is the ROS Contract Language (RCL), a novel Domain Specific Language (DSL) for writing FOL contracts for ROS systems; and Vanda, which is a novel prototype tool that parses RCL and synthesises the runtime monitors.

In our remote inspection case study (§4) we specify contracts for an extant system, then use formal and non-formal verification techniques. Our case study is designed to simply illustrate the core concepts of our approach: FOL contracts, combined with a calculus, guiding the most suitable verification method for each node.

The benefit of letting the contract *guide* the verification is that it is effective even if the specification language used to verify a node does not directly implement FOL. For example, the verification of the **Agent** node (§4.4.3) used program model checking of properties written in Linear-time Temporal Logic (LTL) (which is inherently similar to FOL) enriched with Belief-Desire-Intention (BDI) concepts [68]. However, the other nodes were verified using a variety of techniques, both formal and non-formal. The verification of the **Radiation Sensor** (§4.4.4) uses Hoare Logic, so the contract's FOL properties are easy to represent. The final two nodes were verified using testing/experimental approaches, so the contracts are a guide for the properties to be checked.

Our approach to the case study was to write contracts for, and verify the nodes of, a pre-existing robotic system. However, if we reverse this workflow, the RCL contracts could form a contract-based development approach for building new robotic systems. Used in this way, the contracts would link the system's requirements to its low-level design. RCL could be used to specify the assumptions and guarantees of the proposed system's nodes, and how their inputs and outputs connect the nodes together. Once this version of the design was checked using the calculus, the ROS topic information could be added. These completed contracts could be used to drive a low-level software design, as the starting point for an implementation. Interesting future work could involve extending Vanda to produce message type definitions and skeleton code for ROS nodes, based on the specification in the RCL contracts.

Our approach benefits from three main assumptions or restrictions that nonetheless limit the effectiveness of our current methodology. First, the expressiveness of the streams that link our contracts together is limited by the assumption that each contract consumes one data element from the input stream and produces (at most) one element on the output stream. This means they are limited in handling bursts of data, asynchronous communications, or compound inputs and outputs. We note that the approach can be generalised to improve its expressivity, for example by pre-processing the stream to match the contracts or by splitting the contracts to match the types of data on the streams. Second, our calculus for combining the contracts is currently manually applied and assumes that outputs persist once generated to ensure that all of a contract's inputs are available at the same time. Removing this assumption and mechanising the calculus is left as future work. Finally, we only synthesise Runtime Verification (RV) monitors for a contract's guarantees, not its assumptions. We use our calculus to check that the assumptions of a contract follow the guarantees of the contract(s) that provide its inputs, therefore we chose to monitor only the guarantees to reduce the RV overhead. However, this means that our monitors cannot differentiate between the violation of a guarantee and the violation of an assumption. There is nothing methodologically preventing our tool from synthesising assumption monitors as well, and this extension would enable the RV to detect (for example) where the environment is operating outside of the design-time expectations.

As future work, we have identified three improvements to RCL and Vanda. First, we will investigate adding notation to RCL for specifying that an output is triggered by an input, which would allow

contracts to specify some ordering of events. RCL could also be updated to enable more sophisticated specifications, for example involving real-time constraints or uncertainty. Second, we intend to add support for calculating a contract's assumptions when composing two contracts. For this we would draw inspiration from work by Cobleigh et al. [77].

The third improvement to RCL and VANDA is extending them to handle ROS services and actions. We will update RCL to provide support for linking an input or output to a service or action, and VANDA will be updated to enable services or actions to be monitored. One route to achieving this is to build on our existing approaches use of ROSMonitoring. ROSMontoring 2.0 [58] can monitor services by interceding between the client and server. Given that ROS actions are built on topics and services, an extension that combines these two approaches seems feasible. Another route to monitoring is via introspection, which ROS2 provides for both services[14] and actions[15] This would allow the monitor to read the internal state of the service or action, though this would require the the client to be configured to allow introspection.

It would also be useful to investigate the scalability of our approach; so we intend to apply it to a larger and more complex system, expanding the calculus to cater to other arrangements of nodes where needed. Finally, we intend to explore the level of confidence we can have in a system that has been verified using a mixture of methods; particularly how confidence can be calculated for more complex systems with loops in the information flow.

References

[1] H. Hastie, K. Lohan, M. Chantler, D. A. Robb, S. Ramamoorthy, R. Petrick, S. Vijayakumar, D. Lane, The ORCA Hub: Explainable offshore robotics through intelligent interfaces, in: Explainable Robotic Systems Workshop, ACM Human-Robot Interaction conference„ 2018, p. 2. URL https://arxiv.org/abs/1803.02100

[2] R. Bogue, Robots in the Nuclear Industry: A review of technologies and applications, Industrial Robot: An International Journal 38 (2) (2011) 113–118. doi:10.1108/01439911111106327.

[3] J. M. Aitken, S. M. Veres, A. Shaukat, Y. Gao, E. Cucco, L. A. Dennis, M. Fisher, J. A. Kuo, T. Robinson, P. E. Mort, Autonomous Nuclear Waste Management, IEEE Intelligent Systems 33 (6) (2018) 47–55. doi:10.1109/MIS.2018.111144814.

[4] B. H. Wilcox, Robotic vehicles for planetary exploration, Applied Intelligence 2 (2) (1992) 181–193. doi:10.1007/BF00058762.

[5] A. Flores-Abad, O. Ma, K. Pham, S. Ulrich, A review of space robotics technologies for on-orbit servicing, Progress in Aerospace Sciences 68 (2014) 1–26. doi:10.1016/j.paerosci.2014.03.002.

[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Ng, ROS: an open-source Robot Operating System, in: Workshop on Open Source Software, IEEE, 2009.

[7] M. Foughali, Formal Verification of the Functional Layer of Robotic and Autonomous Systems, Theses, Institut national des sciences appliquées de Toulouse (Dec. 2018). URL https://hal.laas.fr/tel-02080063

[8] S. Fleury, M. Herrb, R. Chatila, GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture, in: International Conference on Intelligent Robots and Systems, IEEE, 1997, pp. 842–849. doi:10.1109/IROS.1997.655108.

[9] R. Halder, J. Proença, N. Macedo, A. Santos, Formal Verification of ROS-Based Robotic Applications Using Timed-Automata, in: Workshop on Formal Methods in Software Engineering, IEEE, 2017. doi:10.1109/FormaliSE.2017.9.

[14] ROS2 Service Introspection: https://docs.ros.org/en/ros2_documentation/kilted/Tutorials/Demos/Service-Introspection.html.

[15] ROS2 Action Introspection: https://docs.ros.org/en/ros2_documentation/kilted/Tutorials/Demos/Action-Introspection.html.

[10] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, M. Fisher, Formal Specification and Verification of Autonomous Robotic Systems: A Survey, ACM Computing Surveys 52 (5) (2019) 1–41. doi:10.1145/3342355.

[11] E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT press, 1999.

[12] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Methods Program. 78 (5) (2009) 293–303. doi:10.1016/j.jlap.2008.08.004.

[13] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions, Springer, 2013. doi:10.1007/978-3-662-07964-5.

[14] M. Farrell, M. Luckcuck, M. Fisher, Robotics and Integrated Formal Methods: Necessity Meets Opportunity, in: Integrated Formal Methods, Vol. 11023 of LNCS, Springer, 2018, pp. 161–171. doi:10.1007/978-3-319-98938-9_10.

[15] C. B. Jones, Tentative Steps Toward a Development Method for Interfering Programs, ACM Trans. on Programming Languages and Systems 5 (4) (1983) 596–619.

[16] B. Meyer, Applying "Design by Contract", Computer 25 (10) (1992) 40–51. doi:10.1109/2.161279.

[17] M. Fisher, An Introduction to Practical Formal Methods Using Temporal Logic, John Wiley & Sons, 2011. doi:10.1002/9781119991472.

[18] R. C. Cardoso, L. A. Dennis, M. Farrell, M. Fisher, M. Luckcuck, Towards Compositional Verification for Modular Robotic Systems, in: Proc. Second Workshop on Formal Methods for Autonomous Systems (FMAS2020), Vol. 329, Electronic Proceedings in Theoretical Computer Science, 2020, pp. 15–22.
URL http://arxiv.org/abs/2012.01648v1

[19] R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando, M. Fisher, Heterogeneous Verification of an Autonomous Curiosity Rover, in: Proc. NASA Formal Methods, Vol. 12229 of LNCS, Springer, 2020, pp. 353–360. doi:10.1007/978-3-030-55754-6_20.

[20] N. Rouquette, I. Incer, A. Pinto, Early design exploration of space system scenarios using assume-guarantee contracts, in: International Conference on Space Mission Challenges for Information Technology, IEEE, 2023, pp. 15–24.

[21] M. Caporuscio, P. Inverardi, P. Pelliccione, Compositional verification of middleware-based software architecture descriptions, in: Proceedings. 26th International Conference on Software Engineering, IEEE, 2004, pp. 221–230.

[22] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-b. Raclet, P. Reinkemeier, A. Sangiovanni-vincentelli, W. Damm, T. Henzinger, K. G. Larsen, Contracts for Systems Design, Tech. Rep. RR-8147, INRIA (2012).
URL https://hal.inria.fr/hal-00757488v1

[23] J. Li, P. Nuzzo, A. Sangiovanni-Vincentelli, Y. Xi, D. Li, Stochastic contracts for cyber-physical system design under probabilistic requirements, in: International Conference on Formal Methods and Models for System Design, no. ii in MEMOCODE '17, ACM Press, New York, USA, 2017, pp. 5–14. doi:10.1145/3127041.3127045.

[24] A. Cimatti, M. Dorigatti, S. Tonetta, Ocra: A tool for checking the refinement of temporal contracts, in: International Conference on Automated Software Engineering (ASE), IEEE, 2013, pp. 702–705. doi:10.1109/ASE.2013.6693137.

[25] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, L. Sha, Compositional verification of architectural models, in: NASA Formal Methods Symposium, Vol. 7226 of LNCS, Springer, 2012, pp. 126–140. doi:10.1007/978-3-642-28891-3_13.

[26] A. Champion, A. Gurfinkel, T. Kahsai, C. Tinelli, Cocospec: A mode-aware contract language for reactive systems, in: Software Engineering and Formal Methods, Vol. 9763 of LNCS, Springer, 2016, pp. 347–366. doi:10.1007/978-3-319-41591-8_24.

[27] A. Champion, A. Mebsout, C. Sticksel, C. Tinelli, The kind 2 model checker, in: Proc. Computer Aided Verification, Vol. 9780 of LNCS, Springer, 2016, pp. 510–517. doi:10.1007/978-3-319-41540-6_29.

[28] H. Bourbouh, M. Farrell, A. Mavridou, I. Sljivo, G. Brat, L. A. Dennis, M. Fisher, Integrating formal verification and assurance: An inspection rover case study, in: NASA Formal Methods Symposium, Vol. 12673 of LNCS, Springer, 2021, pp. 53–71. doi:10.1007/978-3-030-76384-8_4.

[29] M. Broy, A Logical Approach to Systems Engineering Artifacts: Semantic Relationships and Dependencies beyond Traceability – from requirements to functional and architectural views, Software and System Modeling 17 (2) (2018) 365–393. doi:10.1007/s10270-017-0619-4.

[30] I. Ruchkin, J. Sunshine, G. Iraci, B. Schmerl, D. Garlan, IPL: An integration property language for multi-model cyber-physical systems, in: Formal Methods, Vol. 10951 of LNCS, Springer, 2018, pp. 165–184. doi:10.1007/978-3-319-95582-7_10.

[31] L. Baresi, C. Ghezzi, L. Mottola, Loupe: Verifying publish-subscribe architectures with a magnifying lens, IEEE Transactions on Software Engineering 37 (2) (2010) 228–246.

[32] A. Basu, M. Gallien, C. Lesire, T. Nguyen, S. Bensalem, F. Ingrand, J. Sifakis, Incremental component-based construction and verification of a robotic system, in: M. Ghallab, C. D. Spyropoulos, N. Fakotakis, N. M. Avouris (Eds.), ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, Vol. 178 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 631–635. doi:10.3233/978-1-58603-891-5-631.
URL https://doi.org/10.3233/978-1-58603-891-5-631

[33] S. Spellini, M. Lora, F. Fummi, S. Chattopadhyay, Compositional design of multi-robot systems control software on ros, ACM Trans. Embed. Comput. Syst. 18 (5s) (oct 2019). doi:10.1145/3358197.
URL https://doi.org/10.1145/3358197

[34] A. Desai, S. Qadeer, S. A. Seshia, Programming safe robotics systems: Challenges and advances, in: Proc. 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Vol. 11245 of LNCS, Springer, Berlin, Heidelberg, 2018, p. 103–119. doi:10.1007/978-3-030-03421-4_8.

[35] S. Adam, M. Larsen, K. Jensen, U. Schultz, Rule-based dynamic safety monitoring for mobile robots, Journal of Software Engineering for Robotics 7 (2016) 120–141. doi:10.6092/JOSER_2016_07_01_P120.

[36] S. García, P. Pelliccione, C. Menghi, T. Berger, T. Bures, Promise: High-level mission specification for multiple robots, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, ICSE '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 5–8. doi:10.1145/3377812.3382143.
URL https://doi.org/10.1145/3377812.3382143

[37] R. Carvalho, A. Cunha, N. Macedo, A. Santos, Verification of system-wide safety properties of ros applications, in: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2020, pp. 7249–7254. doi:10.1109/IROS45743.2020.9341085.

[38] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, D. Kuperberg, Lightweight specification and analysis of dynamic systems with rich configurations, in: Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 2016, p. 373–383. doi:10.1145/2950290.2950318.

[39] A. Santos, A. Cunha, N. Macedo, C. Lourenço, A framework for quality assessment of ros repositories, in: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016, pp. 4491–4496. doi:10.1109/IROS.2016.7759661.

[40] A. Desai, S. Ghosh, S. A. Seshia, N. Shankar, A. Tiwari, Soter: A runtime assurance framework for programming safe robotics systems, in: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 138–150. doi:10.1109/DSN.2019.00027.

[41] B. Wesselink, K. de Vos, I. Kuertev, M. Reniers, E. Torta, RoboSC: a domain-specific language for supervisory controller synthesis of ROS applications, in: 2023 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2023. doi:10.1109/icra48891.2023.10161436.
URL https://doi.org/10.1109/icra48891.2023.10161436

[42] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, Robochart: Modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. 18 (5) (2019) 3097–3149. doi:10.1007/s10270-018-00710-z.
URL https://doi.org/10.1007/s10270-018-00710-z

[43] M. Farrell, N. Mavrakis, A. Ferrando, C. Dixon, Y. Gao, Formal modelling and runtime verification of autonomous grasping for active debris removal, Frontiers in Robotics and AI 8 (2021). doi:10.3389/frobt.2021.639282.

[44] I. Gavran, R. Majumdar, I. Saha, Antlab: A multi-robot task server, ACM Trans. Embed. Comput. Syst. 16 (5s) (sep 2017). doi:10.1145/3126513.
URL https://doi.org/10.1145/3126513

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

[46] K. Ueda, Guarded horn clauses, in: Proceedings of the 4th Conference on Logic Programming, Vol. 221 of LNCS, Springer, 1985, pp. 168–179. doi:10.1007/3-540-16479-0_17.
URL https://doi.org/10.1007/3-540-16479-0_17

[47] S. Gregory, Parallel logic programming in PARLOG - the language and its implementation, Addison-Wesley, 1987.

[48] A. Pnueli, The Temporal Logic of Programs, in: Foundations of Computer Science, IEEE, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[49] K. A. Elkader, O. Grumberg, C. S. Păsăreanu, S. Shoham, Automated circular assume-guarantee reasoning, in: Formal Methods, Vol. 9109 of LNCS, Springer, 2015, pp. 23–39. doi:10.1007/978-3-319-19249-9_3.

[50] B. Banieqbal, H. Barringer, Temporal Logic with Fixed Points, in: Temporal Logic in Specification, Vol. 398 of LNCS, Springer, 1987, pp. 62–74. doi:10.1007/3-540-51803-7_22.

[51] C. A. Furia, B. Meyer, S. Velder, Loop invariants: Analysis, classification, and examples, ACM Computing Surveys (CSUR) 46 (3) (2014) 1–51.

[52] L. Kovács, A. Voronkov, First-order theorem proving and vampire, in: International Conference on Computer Aided Verification, Springer, 2013, pp. 1–35.

[53] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, X. Yi, A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability, Computer Science Review 37 (2020) 100270. doi:10.1016/j.cosrev.2020.100270.

[54] M. Luckcuck, M. Fisher, L. Dennis, S. Frost, A. White, D. Styles, Principles for the Development and Assurance of Autonomous Systems for Safe Use in Hazardous Environments (Jun. 2021). doi:10.5281/zenodo.5012322.
URL https://doi.org/10.5281/zenodo.5012322

[55] M. Luckcuck, Using formal methods for autonomous systems: Five recipes for formal verification, Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability 0 (0) (2021) 1748006X2110349. doi:10.1177/1748006X211034970.

[56] K. Y. Rozier, Specification: The biggest bottleneck in formal methods and autonomy, in: Working Conference on Verified Software: Theories, Tools, and Experiments, Vol. 9971 of LNCS, Springer, 2016, pp. 8–26. doi:10.1007/978-3-319-48869-1_2.

[57] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, V. Mascardi, ROSMonitoring: A runtime verification framework for ROS, in: Towards Autonomous Robotic Systems, Vol. 12228 of LNCS, Springer, 2020, pp. 387–399. doi:10.1007/978-3-030-63486-5_40.

[58] M. G. Saadat, A. Ferrando, L. A. Dennis, M. Fisher, Rosmonitoring 2.0: Extending ROS runtime verification to services and ordered topics, in: M. Luckcuck, M. Xu (Eds.), Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024, Vol. 411 of EPTCS, 2024, pp. 38–55. doi:10.4204/EPTCS.411.3.

[59] D. Ancona, L. Franceschini, A. Ferrando, V. Mascardi, RML: Theory and Practice of a Domain Specific Language for Runtime Verification, Science of Computer Programming 205 (2021) 102610. doi:10.1016/j.scico.2021.102610.

[60] T. Wright, A. West, M. Licata, N. Hawes, B. Lennox, Simulating ionising radiation in gazebo for robotic nuclear inspection challenges, Robotics 10 (3) (2021). doi:10.3390/robotics10030086.
URL https://www.mdpi.com/2218-6581/10/3/86

[61] L. A. Dennis, Gwendolen semantics: 2017, Tech. Rep. ULCS-17-001, University of Liverpool, Department of Computer Science (2017).
URL https://intranet.csc.liv.ac.uk/research/techreports/tr2017/ulcs-17-001.pdf

[62] L. Carlone, F. Dellaert, Duality-based verification techniques for 2d slam, in: 2015 IEEE international conference on robotics and automation (ICRA), IEEE, 2015, pp. 4589–4596. doi:10.1109/ICRA.2015.7139835.

[63] L. Carlone, D. M. Rosen, G. Calafiore, J. J. Leonard, F. Dellaert, Lagrangian duality in 3d slam: Verification techniques and optimal solutions, in: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2015, pp. 125–132. doi:10.1109/IROS.2015.7353364.

[64] J. Briales, J. Gonzalez-Jimenez, Fast global optimality verification in 3d slam, in: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 4630–4636. doi:10.1109/IROS.2016.7759681.

[65] Y. Hasegawa, Y. Fujimoto, Experimental verification of path planning with slam, IEEJ Journal of Industry Applications 5 (3) (2016) 253–260. doi:10.1541/ieejjia.5.253.

[66] F. Duchoň, J. Hažík, J. Rodina, M. Tölgyessy, M. Dekan, A. Sojka, Verification of slam methods implemented in ros, Journal of Multidisciplinary Engineering Science and Technology (JMEST) 6 (2019). doi:10.22223/tr.2016-1/2011.

[67] R. Sivỳ, D. Perduková, Verification of slam methods on ros platform, Transactions of the VŠB-Technical University of Ostrava, Mechanical Series 62 (2016) 59–66. doi:10.22223/tr.2016-1/2011.

[68] L. A. Dennis, M. Fisher, M. P. Webster, R. H. Bordini, Model checking agent programming languages, Automated Software Engineering 19 (1) (2012) 5–63. doi:10.1007/s10515-011-0088-x.

[69] L. A. Dennis, The mcapl framework including the agent infrastructure layer and agent java pathfinder, The Journal of Open Source Software 3 (24) (2018).

[70] W. Visser, K. Havelund, G. Brat, S.-J. Park, F. Lerda, Model Checking Programs, Automated Software Engineering 10 (2) (2002) 3–11. doi:10.5555/786768.786967.

[71] A. S. Rao, M. Georgeff, BDI Agents: From Theory to Practice, in: International Conference on Multi-Agent Systems, AAAI, 1995, pp. 312–319.

[72] L. A. Dennis, M. Fisher, Verifiable Autonomous Systems: Using Rational Agents to Provide Assurance about Decisions Made by Machines, Cambridge University Press, 2023.

[73] C. A. R. Hoare, An axiomatic basis for computer programming, Comms. of the ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.

[74] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: Logic for Programming Artificial Intelligence and Reasoning, Vol. 6355 of LNCS, Springer, 2010, pp. 348–370. doi:10.1007/978-3-642-17511-4_20.

[75] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A Modular Reusable Verifier for Object-Oriented Programs, in: Formal Methods for Components and Objects, Vol. 4111 of LNCS, Springer, 2005, pp. 364–387. doi:10.1007/11804192_17.

[76] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 4963 of LNCS, Springer, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[77] J. M. Cobleigh, D. Giannakopoulou, C. S. Pasareanu, Learning Assumptions for Compositional Verification, in: Tools and Algorithms for the Construction and Analysis of Systems, Vol. 2619 of LNCS, Springer, 2003, pp. 331–346. doi:10.1007/3-540-36577-X_24.

A Remote Inspection Contracts

This appendix contains the full listing of the RCL context and the contracts of all four nodes from our case study in §4.

Context

$$
\begin{aligned}
\mathit{WayP}: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{N} \\
\mathit{RadStat}: &\quad \{\mathit{red}, \mathit{orange}, \mathit{green}\} \\
\mathit{CommandSet}: &\quad \{\mathit{move}(\mathbb{R}, \mathbb{R}), \mathit{inspect}(\mathbb{N})\} \\
\mathit{PositionType}: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{B} \\
\mathit{AtType}: &\quad \mathbb{R} \times \mathbb{R} \to \mathbb{B} \\
\mathit{InspectedType}: &\quad \mathbb{N} \to \mathbb{B} \\
\mathit{SensorsType}: &\quad \emptyset
\end{aligned}
$$

Agent

inputs ($\mathit{wayP} : \mathit{WayP}, \mathit{at} : \mathit{AtType}, \mathit{radiationStatus} : \mathit{RadStat}$,
$\mathit{inspected} : \mathit{InspectedType}$)

outputs ($\mathit{command} : \mathit{CommandSet}$)

topics ($\mathit{gazebo_radiation_plugin/Snapshot\ command}$
$\quad \mathit{matches}: \mathit{out.command}$,
$\mathit{gazebo_radiation_plugin/Snapshot\ inspected}$
$\quad \mathit{matches}: \mathit{in.inspected}$,
$\mathit{gazebo_radiation_plugin/Snapshot\ at\ matches}: \mathit{in.at}$,
$\mathit{gazebo_radiation_plugin/Snapshot\ wayPNow}$
$\quad \mathit{matches}: \mathit{in.wayP}$,
$\mathit{gazebo_radiation_plugin/Snapshot\ radiationStatus}$
$\quad \mathit{matches}: \mathit{in.radiationStatus}$)

assume $(\mathit{in.radiationStatus} \in \{\mathit{red}, \mathit{orange}, \mathit{green}\})$

guarantee
$$
\begin{aligned}
&(\forall\, x', y' \in \mathit{REAL}, i \in \mathit{NATURAL} \cdot \mathit{in.at}(x', y') = \mathit{TRUE} \\
&\land\ \mathit{in.wayP}(x', y') = i \land \mathit{in.inspected}(i) = \mathit{TRUE} \\
&\land\ \mathit{in.radiationStatus} \notin \{\mathit{red}, \mathit{orange}\} \\
&\Rightarrow \exists\, x, y \in \mathit{REAL} \cdot (\mathit{in.wayP}(x, y) = i + 1 \\
&\lor\ (\forall\, x'', y'' \in \mathit{REAL} \cdot \mathit{in.wayP}(x'', y'') \neq i + 1 \\
&\land\ \mathit{in.wayP}(x, y) = 0)) \land \mathit{out.command} = \mathit{move}(x, y))
\end{aligned}
$$

guarantee
$$
\begin{aligned}
&(\forall\, x', y' \in \mathit{REAL}, i \in \mathit{NATURAL} \cdot \mathit{in.at}(x', y') = \mathit{TRUE} \\
&\land\ \mathit{in.wayP}(x', y') = i \land \mathit{in.inspected}(i) = \mathit{FALSE} \\
&\Rightarrow \mathit{out.command} = \mathit{inspect}(i))
\end{aligned}
$$

guarantee
$$
\begin{aligned}
&(\forall\, x', y' \in \mathit{REAL}, i \in \mathit{NATURAL} \cdot \\
&\mathit{in.radiationStatus} \in \{\mathit{red}, \mathit{orange}\} \\
&\lor\ \neg\, \exists\, x, y \in \mathit{REAL} \cdot \mathit{in.wayP}(x, y) = i + 1 \\
&\qquad \Rightarrow \exists\, x'', y'' \in \mathit{REAL} \\
&\cdot\ \mathit{in.wayP}(x'', y'') = 0 \land \mathit{out.command} = \mathit{move}(x'', y''))
\end{aligned}
$$

Localisation

inputs	$(sensors : SensorsType)$
outputs	$(position : PositionType)$
topics	$(gazebo_radiation_plugin/Snapshot\ position$ $matches :\ out.position)$
assume	$(TRUE)$
guarantee	$(\exists!\, x, y \in REAL \cdot out.position(x, y))$

Navigation

inputs	$(position : PositionType, command : CommandSet)$
outputs	$(at : AtType)$
topics	$(gazebo_radiation_plugin/Snapshot\ command$ $matches :\ in.command,$ $gazebo_radiation_plugin/Snapshot\ currentLoc$ $matches :\ in.position,$ $gazebo_radiation_plugin/Snapshot\ at\ matches :\ out.at)$
assume	$(\exists!\, x, y \in REAL \cdot in.position(x, y) = TRUE)$
guarantee	$(\forall\, x, y \in REAL \cdot in.command = move(x, y)$ $\wedge\ in.position(x, y) = TRUE \Leftrightarrow out.at(x, y) = TRUE)$

RadiationSensor

inputs	$(r : REAL, command : CommandSet)$
outputs	$(radiationStatus : RadStat, inspected : InspectedType)$
topics	$(gazebo_radiation_plugin/Snapshot\ r\ matches :\ r,$ $gazebo_radiation_plugin/Snapshot\ command$ $matches :\ in.command,$ $gazebo_radiation_plugin/Snapshot\ inspected$ $matches :\ inspected,$ $gazebo_radiation_plugin/Snapshot\ radiationStatus$ $matches :\ out.radiationStatus)$
assume	$(0 \leq in.r)$
guarantee	$(\forall\, i \in NATURAL \cdot in.command = inspect(i) \Rightarrow$ $(out.inspected(i) = TRUE$ $\wedge\ 0 \leq in.r \wedge in.r < 120 \Rightarrow out.radiationStatus = green$ $\wedge\ 120 \leq in.r \wedge in.r < 250 \Rightarrow out.radiationStatus = orange$ $\wedge\ 250 \leq in.r \Rightarrow out.radiationStatus = red))$

B Hoare Proofs

This appendix contains the full Hoare Logic Proofs used in §4.4.4

B.1 Proof 1

$\{radiation_at(i) < 10\}$

```
1 input := radiation_at(i)
```

$\{input < 10\}$ - Assignment Axiom

```
1 IF (input < 10) THEN
```

$\{input < 10 \wedge input < 10\}$
$\{input < 10 \wedge green = green\}$ - strengthening

```
1    output := green
```

$\{input < 10 \wedge output = green\}$ - Assignment Axiom
$\{output = green\}$ - weakening

```
1    ELSE
2
```

$\{input < 10 \wedge input \geq 10\}$
$\{\bot\}$ - strengthening

```
1     IF (input < 20) THEN
```

$\{\bot \wedge input < 20\}$
$\{\bot\}$ - strengthening

```
1       output := orange
```

$\{\bot\}$ - Assignment axiom

```
1     ELSE
2
```

$\{\bot \wedge input \geq 20\}$
$\{\bot\}$ - strengthening

```
1       output := red
```

$\{\bot\}$ - Assignment axiom
$\{output = green\}$ - weakening
$\{output = green\}$ – Conditional Rule
$\{output = green\}$ – Conditional Rule

B.2 Proof 2

$\{10 \leq radiation_at(i) < 20\}$

```
1 input := radiation_at(i)
```

$\{10 \leq input < 20\}$ - Assignment Axiom

```
1 IF (input < 10) THEN
```

$\{10 \leq input < 20 \wedge input < 10\}$
$\{\bot\}$ - strengthening

```
1    output := green
```

$\{\bot\}$ - Assignment Axiom
$\{output = orange\}$ - weakening

```
1    ELSE
2
```

$\{10 \leq input < 20 \wedge input \geq 10\}$
$\{10 \leq input < 20\}$ - strengthening

```
1    IF (input < 20) THEN
```

$\{10 \leq input < 20 \wedge input < 20\}$
$\{10 \leq input < 20 \wedge orange = orange\}$ - strengthening

```
1      output := orange
```

$\{10 \leq input < 20 \wedge output = orange\}$ - Assignment axiom
$\{output = orange\}$ - weakening

```
1    ELSE
2
```

$\{10 \leq input < 20 \wedge input \geq 20\}$
$\{\bot\}$ - strengthening

```
1      output := red
```

$\{\bot\}$ - Assignment axiom
$\{output = orange\}$ - weakening
$\{output = orange\}$ – Conditional Rule
$\{output = orange\}$ – Conditional Rule

B.3 Proof 3

$\{radiation_at(i) \geq 20\}$

```
1 input := radiation_at(i)
```

$\{input \geq 20\}$ - Assignment Axiom

```
1 IF (input < 10) THEN
```

$\{input \geq 20 \wedge input < 10\}$
$\{\bot\}$ - strengthening

```
1    output := green
```

$\{\bot\}$ - Assignment Axiom
$\{output = red\}$ - weakening

```
1    ELSE
2
```

$\{input \geq 20 \wedge input \geq 10\}$
$\{input \geq 20\}$ - strengthening

```
1    IF (input < 20) THEN
```

$\{input \geq 20 \wedge input < 20\}$
$\{\bot\}$ - strengthening

```
1      output := orange
```

$\{\bot\}$ - Assignment axiom
$\{output = red\}$ - weakening

```
1     ELSE
2
```

$\{input \geq 20 \wedge input \geq 20\}$
$\{input \geq 20 \wedge red = red\}$ - strengthening

```
1       output := red
```

$\{input \geq 20 \wedge output = red\}$ - Assignment axiom
$\{output = red\}$ - weakening
$\{output = red\}$ – Conditional Rule
$\{output = red\}$ – Conditional Rule

B.4 Proof 4

$\{True\}$
$\{radiation_at(i) = radiation_at(i)\}$ - strengthening

```
1 input := radiation_at(i)
```

$\{input = radiation_at(i)\}$ - Assignment Axiom

```
1 IF (input < 10) THEN
```

$\{input = radiation_at(i) \wedge input < 10\}$

```
1    output := green
```

$\{input = radiation_at(i) \wedge input < 10\}$ - Assignment Axiom
$\{input = radiation_at(i)\}$ - weakening

```
1    ELSE
2
```

$\{input = radiation_at(i) \wedge input \geq 10\}$
$\{input = radiation_at(i)\}$ - strengthening

```
1     IF (input < 20) THEN
```

$\{input = radiation_at(i) \wedge input < 20\}$

```
1       output := orange
```

$\{input = radiation_at(i) \wedge input < 20\}$ - Assignment axiom
$\{input = radiation_at(i)\}$ - weakening

```
1     ELSE
2
```

$\{input = radiation_at(i) \wedge input \geq 20\}$

```
1       output := red
```

$\{input = radiation_at(i) \wedge input \geq 20\}$ - Assignment axiom
$\{input = radiation_at(i)\}$ - weakening
$\{input = radiation_at(i)\}$ – Conditional Rule
$\{input = radiation_at(i)\}$ – Conditional Rule